Institutional Repository - Research Portal
Dépôt Institutionnel - Portail de la Recherche
researchportal.unamur.be
University of Namur

THESIS / THÈSE

MASTER EN SCIENCES INFORMATIQUES

Étude critique d'un outil d'aide à la dérivation automatique de procédures PROLOG

Guebel, Benoît

Award date:
1992

Awarding institution:
Universite de Namur

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal ?

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 05. May. 2023

Facultés Universitaires Notre-Dame de la Paix
Institut d'Informatique

Rue Grangagnage, 21, B-5000 NAMUR (Belgium)

ETUDE CRITIQUE D'UN OUTIL D'AIDE A LA DERIVATION AUTOMATIQUE DE PROCEDURES PROLOG

Benoît GUEBEL

Promoteur: Professeur Baudouin LE CHARLIER

Mémoire présenté en vue
de l'obtention du titre de
Licencié et Maître en Informatique

Année académique 1991-92

J'exprime toute ma gratitude à Monsieur Baudouin Le Charlier, promoteur de ce mémoire, pour ses remarques judicieuses et ses conseils précieux. Je le remercie pour l'amabilité avec laquelle il m'a toujours reçu.

Mes très sincères remerciements vont également à Monsieur Pierre De Boeck sans qui ce mémoire ne serait pas ce qu'il est. Je le remercie tout particulièrement pour son aide, ses commentaires et ses idées qui me guidèrent d'un bout à l'autre dans la réalisation de ce mémoire.

Je ne terminerai pas sans remercier toutes les personnes qui m'ont soutenu, de près ou de loin, pour réaliser ce mémoire.

A Fabienne et Justine.

Résumé

Pour s'assurer de la correction de procédures PROLOG,un outil d'analyse statique de types a été développé par les chercheurs du projet FOLON, à l'Institut d'Informatique de Namur. Grâce à lui, il est possible de déduire des informations concernant les types et les modes des paramètres à chaque point d'exécution d'une procédure PROLOG. De telles propriétés permettent de détecter statiquement des appels incorrects. Le présent mémoire tente d'étudier de façon critique cet outil par l'analyse approfondie d'exemples et de tirer les conclusions de son utilisation.

Abstract

A static type analysis tool for ensuring correctness of PROLOG procedures was developed by the researchers of the FOLON project at the Institute of Computer Science of the University of Namur. With this tool, type properties can be deduced at each execution point of a PROLOG procedure. Such properties allow to detect inconsistent calls at compile-time. The aim of this thesis is to evaluate this tool by performing detailed analysis of examples and then to draw conclusions from these experiments.

0. INTRODUCTION.

Programmer en PROLOG n'est pas programmer purement en logique. Il existe en effet un important fossé entre la sémantique déclarative et la sémantique procédurale d'un programme PROLOG. Les différences entre ces deux sémantiques sont à l'origine de la méthodologie proposée par Yves Deville. Celui-ci considère en effet que tout développement d'un programme logique se décompose en trois étapes:

1. Elaboration d'une spécification: cette étape consiste essentiellement en la description de la relation à implémenter par la procédure ainsi que son domaine; des informations sur les conditions d'utilisation de la procédure sont également présentes.
2. Construction d'une description logique correcte par rapport à la spécification: formalisation en logique de premier ordre de la relation et de son domaine.
3. Dérivation d'une procédure Prolog correcte (et efficace) à partir de la description logique . C'est seulement lors de cette étape que seront traités les aspects non logiques propres à Prolog.

La troisième étape peut elle-même être divisée en trois étapes:

1. Dérivation syntaxique de la description logique correcte en PROLOG pur (sans négation)
2. Dérivation d'une procédure PROLOG correcte à partir des résultats de l'étape précédente; cela consiste principalement en la vérification de critères de types et de modes des paramètres et occasionnellement en transformations telles que le réarrangement des littéraux dans une clause, l'ajoût de littéraux de type checking, ...
3. Transformation de la procédure correcte en une plus efficace, tout en la gardant correcte. Dans cette étape, des actions telles que l'ajoût d'information de contrôle et d'évaluation partielle sont effectuées.

Un analyseur a été développé pour effectuer la deuxième de ces étapes de façon semi-automatique.

Dans ce mémoire, nous allons procéder à une étude critique de cet outil grâce à des tests sur des procédures PROLOG classiques (`efface/3`, `reverse/2`, ...).et sur une procédure issue de la réalisation d'une application simple de base de données. Nous allons tâcher de vérifier la correction, la fiabilité et l'efficacité de cet outil. Fournit-il les réponses que nous attendons et si non, pourquoi? Fournit-il d'autres réponses? Si oui, sont-elles correctes?
Est-il suffisamment convivial? Ses temps de réponse sont-ils aceptables? Voilà d'autres points à débattre.

Le plan du mémoire découle de ces motivations. Afin d'aborder l'étude de cet outil, quelques notions fondamentales de la logique des prédicats et de la programmation logique seront rappelés dans le chapitre 1. Dans le chapitre 2, nous rappellerons la méthodologie de Yves Deville: un processus de construction de procédures PROLOG en trois étapes. Le chapitre 3 s'attachera plus particulièrement à la dérivation d'une procédure PROLOG correcte à partir d'une description logique correcte. Dans le chapitre 4, après avoir rappelé la notion d'analyse statique de clauses PROLOG, nous présenterons l'analyseur ainsi que les outils développés pour faciliter l'analyse de clauses. Cet analyseur sera ensuite testé dans le chapitre 5 grâce à des procédures PROLOG classiques et par l'étude d'une application simple de base de données. Dans le chapitre 6, nous présenterons quelques extensions possibles et souhaitées, découvertes lors de l'utilisation de cet outil. Le chapitre 7 servira de conclusion à ce mémoire.

1. RAPPEL DE CONCEPTS IMPORTANTS EN PROGRAMMATION PROLOG.

Avant d'aborder le sujet du mémoire, il est utile de procéder à quelques rappels de notions fondamentales. Les concepts présentés ici ne seront que brièvement évoqués. Pour des explications plus détaillées, il est conseillé au lecteur de se référer aux maîtres du genre Lloyd [6], Kowalski [5], Clocksin [3].Il est même conseillé à l'habitué de la programmation logique de ne pas s'attarder sur ce chapitre.

1.1. CONCEPTS DE LA SEMANTIQUE DECLARATIVE.

1.1.1. LOGIQUE DU PREMIER ORDRE.

La théorie du premier ordre comprend un alphabet, un langage du premier ordre, un ensemble d'axiomes et un ensemble de règles d'inférence qui permettent de déduire des conséquences logiques à partir de ces axiomes et d'autres formules logiques.L'alphabet est constitué par tous les symboles qu'on peut écrire dans le langage du premier ordre.

Définition 1.1. (alphabet)

Un alphabet consiste en sept types de symboles :
1) Les variables
2) Les constantes
3) Les symboles de fonctions
4) Les symboles de prédicats
5) Les connecteurs logiques: $\sim$, $\Rightarrow$, $\Leftrightarrow$, $\vee$ et $\wedge$.
6) Les quantificateurs
7) Les symboles de ponctuation: '(', ')', ','.

Dans les langages de programmation logique, le terme constitue la structure de données de base. Un terme est défini récursivement par:

Définition 1.2. (terme)

- Une variable est un terme.
- Une constante est un terme.
- Si `f` est un symbole de fonction d'arité `n` et `t1, . . . , tn` sont des termes, alors `f(t1, ... , tn)` est aussi un terme.

Définition 1.3. (formule bien formée)

Une formule bien formée (notée *fbf*) est définie inductivement comme suit:
- Si `p` est un symbole de prédicat d'arité `n` et `t1, . . . , tn` sont des termes alors `p(t1, ... ,tn)` est aussi une *fbf* appelée atome.
- Si `F` et `G` sont deux formules bien formées alors $\sim$`F`, `F` $\wedge$ `G`, `F` $\vee$ `G`, `F` $\Rightarrow$ `G`, `F` $\Leftrightarrow$ `G` sont aussi des *fbfs*.

- Si F est une *fbf* et X est une variable, alors ∀ X et ∃ X sont des *fbfs*.

Définition 1.4. (fbf fermée)

Une formule bien formée est dite fermée si toutes ses variables sont quantifiées universellement ou existentiellement.

Définition 1.5. (littéral)

Un littéral est soit un atome soit la négation d'un atome.

Définition 1.6. (clause)

Une clause est une formule de la forme A ← B1, ... , Bn où toutes les variables sont quantifiées universellement et où les virgules dans le second membre dénotent une conjonction. Le symbole ← dénote l'implication logique.
A est appelé tête de clause, B1,... , Bn est appelé corps de la clause.

Définition 1.7. (but)

Un but est une formule de la forme : ← B1, ... , Br
C'est une clause qui a une tête vide.

Définition 1.8. (clause vide)

Une clause dont la tête et le corps sont vides est appelée clause vide ou contradiction et est notée ⊕.

Définition 1.9. (programme logique)

Un programme logique est un ensemble fini de clauses.

1.1.2. INTERPRETATION ET MODELE.

Dans cette section, nous allons nous attacher à l'aspect sémantique des programmes logiques en fixant une signification aux symboles et formules du langage. Autrement dit, nous allons définir une interprétation des formules des programmes. Une interprétation consiste à définir un domaine de valeurs pour les constantes, les variables, les symboles de fonction et les symboles de prédicat. Une interprétation d'un programme logique, ou d'un ensemble de formules logiques en général, est une manière de fixer un domaine de discours pour les symboles du langage. La définition d'une interprétation permet de parler de la valeur de vérité des formules. La valeur de vérité d'une formule est définie de la façon suivante:
Si la formule se réduit à un atome p(x1, ... , xn), alors sa valeur de vérité est fixée par l'interprétation; sinon c'est une formule construite à partir de connecteurs et de quantificateurs logiques; sa valeur de vérité est obtenue par les tables de vérité de la façon habituelle.
Etant donné un programme logique, il existe certaines interprétations où toutes les clauses de ce programme ont une valeur de vérité VRAIE. De telles interprétations sont appelées modèles.

Définition 1.10. (conséquence logique)

Soit S = {F1, ..., Fn} un ensemble de formules fermées d'un langage du premier ordre. On dit que F est une conséquence logique (|=) de S si et seulement si, tout modèle de S est aussi un

modèle de F.

Définition 1.11. (Univers de Herbrand)

Soit L un langage du premier ordre, l'Univers d'Herbrand est l'ensemble de tous les termes ground (c-à-d les termes ne contenant pas de variables),ces termes étant construits à partir des constantes et des symboles de fonctions du langage.

Définition 1.12. (base de Herbrand)

Une base d'Herbrand associée à un langage L est l'ensemble des atomes ground .

1.2. CONCEPTS DE LA SEMANTIQUE PROCEDURALE

1.2.1. DEFINITIONS PRELIMINAIRES.

Définition 1.13. (substitution)

Une substitution est un ensemble fini de la forme $\theta = \{ x1/ t1, \ldots, xn/tn \}$ où les xi sont des variables distinctes et les ti des termes. Chaque ti est distinct de xi. Chaque élément xi / ti est appelé liaison. Si ti est ground pour tout i, alors la substitution θ est dite ground.

Définition 1.14. (expression)

Une expression est soit un terme, un littéral ou une conjonction ou disjonction de littéraux.

Définition 1.15. (instance)

Soit E une expression et soit $\theta = \{ x1/ t1, \ldots, xn / tn \}$ une substitution. Eθ est l'expression obtenue en remplaçant simultanément chaque occurrence de la variable xi par le terme ti (i =1 , ... , n). Eθ est appelé instance de E par θ.

Définition 1.16. (composition de substitutions)

Soient $\theta = \{ u1/ s1, \ldots, un / sn \}$ et $\sigma = \{ v1/ t1, \ldots, vm/ tm \}$ deux substitutions. La composition $\theta\sigma$ des deux substitutions θ et σ est la substitution définie par l'ensemble: $\{u1/s1\sigma, \ldots, um/ sm\ \sigma, v1/ t1, \ldots, vn/ tn \}$ en supprimant tout élément tel que ui = siσ et tout élément vi / ti tel que $vi \in \{ u1, \ldots, un \}$.
La composition de deux substitutions θ et σ est la substitution γ dont l'application sur toute expression E a le même effet que l'application de la substitution θ à l'expression E suivie de l'application de la substitution σ à l'expression Eθ.

Définition 1.17. (unification)

Soient E1, E2 deux expressions. Soit θ une substitution. On dit que θ unifie E1 et E2 si et seulement si les deux expressions E1 et E2 sont syntaxiquement identiques, symbole par symbole.

Définition 1.18. (most general unifier)

Soient E1, E2 deux expressions. Soit θ une substitution. On dit que θ est un mgu (most general

unifier) de E1 et E2 si et seulement si θ unifie E1 et E2 et pour toute substitution σ qui unifie E1 et E2 on a :
∃ γ une substitution telle que σ = θγ.

L'unification est l'opération de base en programmation logique de façon analogue à l'instruction d'affectation dans les langages conventionnels.

1.2.2. SLD-DERIVATION

Soit P un programme logique, G le but ← B1, ... , Bq , ... , Bret C la clause A ← C1, ... , Cn.
Le but G′ dérivé de G en sélectionnant le littéral Bq et la clause C est le but :
(← B1, ..., Bq -1 , C1, ... , Cn , Bq +1 , ... , Br) θ
où θ est un mgu de A et Bq .
Bq est appelé le littéral sélectionné.

Soit P un programme logique, et G un but. La SLD-dérivation de P ∪ {G} est constituée d'une séquence de substitutions q1, ... , qn et d'une séquence de buts G0 = G, G1 , ... , Gn dans lesquels Gk +1 est dérivé de Gk et de la clause Ck +1 en utilisant la substitution θ.

Une SLD-refutation de P ∪ {G} est une SLD-dérivation dans laquelle le dernier but est la clause vide Gn = ◊ .

Expliquons à présent comment l'interpréteur prouve qu'une formule:

∃ y1, ... , yn (B1 ∧ ... ∧ Br)

est une conséquence logique d'un programme P. Pour prouver ce but, l'interpréteur travaille par réfutation. D'abord, on nie le but qu'on veut prouver, puis on le rajoute aux clauses du programme et on essaye d'aboutir à une contradiction c-à-d la clause vide. Donc, partant du but:

← B1, ... , Br

on dérive à chaque étape un nouveau but en sélectionnant un littéral dans ce but et en le remplaçant par le corps d'une clause dont la tête s'unifie avec ce littéral. Certaines clauses sont des faits c-à-d leur corps est vide. Donc on peut progressivement diminuer le nombre de littéraux dans le but qu'on veut prouver.

Les négations sont considérées de la manière suivante (négation par échec):
not (p) réussit (c-à-d on réussit à prouver que p est vraie) si p échoue et inversement.

1.2.3. ARBRE DE DERIVATION.

Le comportement de l'interpréteur peut être expliqué par la construction d'un arbre appelé arbre de dérivation.

Définition 1.19. (règle de sélection)

Une règle de sélection de but est une fonction de l'ensemble des buts vers l'ensemble des littéraux. Cette fonction associe à chaque but

← B1, ... , Br

un littéral Bq ∈ { B1 , ... , Br }. Bq est appelé le sous-but sélectionné.

Définition 1.20. (arbre de dérivation)

Soit P un programme, G un but et R une règle de sélection de buts. L'arbre de dérivation de P ∪ { G }, en utilisant la règle de sélection R, est défini comme suit:
(1) chaque nœud de l'arbre est un but.
(2) une substitution est associée à chaque arc de l'arbre.
(3) le nœud racine est le but initial G.
(4) les nœuds qui sont des buts vides n'ont pas de descendants. Ces nœuds sont appelés nœuds de succès. La branche de l'arbre partant du nœud racine et aboutissant à un nœud succès est appelée branche de succès.
(5) soit ← L1 , ... , Lk , ... , Ln un nœud G′ de l'arbre de dérivation (n ≥ 1) et Lk le littéral sélectionné par la règle de sélection R.

(a) si Lk est un littéral positif c-à-d un littéral qui n'est pas une négation, alors :

(i) ce nœud possède un nœud descendant pour chaque clause

A ← B1, ... , Br

dont la tête s'unifie avec Lk .

(ii) le mgu θ tel que Aθ = Lkθ est attaché à l'arc.

(iii) le nœud descendant est

← (L1, ... ,Lk −1, B1, ... , Br , Lk +1 , ... , Ln) θ

si le nœud n'a pas de descendant, ce nœud est appelé nœud échec et la branche reliant le nœud racine à ce nœud est appelée branche d'échec.

(b) si Lk est une négation not (Ak) , alors, ou bien :

(i) l'arbre de dérivation de P ∪ {← Ak } est un arbre d'échec. Dans un tel cas, l'unique nœud descendant est : ← L1, ... , Lk −1 , Lk +1 , ... , Ln et la substitution attachée à cet arc est la substitution identité.

(ii) il existe une branche de succès dans l'arbre de résolution de P ∪ {← Ak } via R. Dans ce cas il n'y a pas de nœud descendant pour G. G′ est un nœud d'échec et toute branche partant du nœud racine et aboutissant au nœud G′ est une branche d'échec.

(iii) L'unique descendant pour le nœud G′ est lui-même et la substitution attachée à cet arc est la substitution identité. On obtient donc une branche infinie.

Un arbre de dérivation est un arbre d'échec si et seulement si, toute branche de cet arbre est une branche d'échec.

Une explication plus détaillée de l'arbre de dérivation peut être trouvée notamment dans [1] et [6].

Définition 1.21. (réponse calculée)

Soit P un programme logique et G un but. La substitution θ est une réponse calculée (c.a.s.) de P ∪ { G } si et seulement s'il existe dans l'arbre de dérivation de P ∪ { G } une branche de succès telle que θ soit la composition des substitutions attachées aux arcs de cette branche.

2. CONSTRUCTION DE PROCEDURES PROLOG: UN PROCESSUS EN TROIS ETAPES.

Nous décrivons dans ce chapitre les trois étapes importantes de la méthodologie de Deville [1] pour construire des procédures Prolog à partir de leur spécification. Nous rappelons d'abord au moyen de quelques exemples l'important fossé existant entre la sémantique déclarative et la sémantique procédurale d'un programme Prolog. En effet, les différences entre ces deux sémantiques sont à l'origine de la méthodologie.L'exposé est un résumé des chapitres 2, 4, 5, 8, 9 de Deville [1].

2.1. DIFFERENCES ENTRE SEMANTIQUE DECLARATIVE ET SEMANTIQUE PROCEDURALE

La sémantique déclarative d'un programme Prolog est l'ensemble des conséquences logiques du programme, ce dernier étant vu comme une théorie logique (un ensemble de clauses de Horn); sa sémantique procédurale est l'ensemble des théorèmes qui peuvent être prouvés à partir du moteur d'inférence de Prolog (SLDNF-resolution).
Prolog serait purement logique ssi ces deux sémantiques étaient parfaitement équivalentes, c-à-d:

- toute conséquence logique peut être prouvée par Prolog; on parle alors de *completeness*.
- tout théorème prouvé par Prolog est une conséquence logique; on parle alors de *soundness*.

Cette équivalence, si elle est vraie avec un Prolog "pur" (sans négation, sans built-in extra-logique, avec breadth-first search, ...) ne tient plus dès que l'on étend ce Prolog "pur" avec le minimum nécessaire (négation, built-in extra-logique, depth-first search) pour en faire un langage permettant une programmation raisonnablement efficace et aisée. Les exemples qui suivent illustrent aussi bien des cas d'uncompleteness que d'unsoundness.
Nous conseillons au lecteur intéressé de se referer à l'ouvrage de Lloyd [6] pour un traitement rigoureux et complet de ces deux sémantiques et de leurs différences.

- **uncompleteness**

Certaines conséquences logiques ne seront jamais trouvées par Prolog. Cette incomplétude est due principalement à la manière dont Prolog parcourt l'arbre de preuve(depth-first search). La négation peut être une autre raison de cette incomplétude ainsi que l'utilisation de certains built-ins "extra-logiques".

Exemple 2.1. (depth-first search)

```
p ← p.
p.
```

Prolog ne peut pas prouver que `p` est vrai (parcours d'une branche infinie), alors que `p` est une conséquence logique de ce programme.

Exemple 2.2. (négation)

```
t ← p.
t ← not(p).
p ← p.
```

`t` est une conséquence logique de ce programme mais `t` ne peut être prouvé vrai par la SLDNF-resolution; cela demande en effet de prouver soit que `p` est vrai ou soit que `not (p)` est vrai, c-à-d que `p` ne peut être prouvé vrai (négation par échec). Dans les deux cas et quelle que soit la règle de recherche, on parcourera une branche infinie.

Exemple 2.3. *(cut)*

```
p(Z)← !,x=a.
p(b).
```

`p(b)` est une conséquence loqique mais la présence du cut empêchera pourtant Prolog de le prouver.

Exemple 2.4. *(built-in arithmétique)*

```
min(X,Y,X)← X ≤ Y.
min(X,Y,Y)← Y ≤ X.
```

Le goal `min(X,Y,4)` est bien une conséquence logique (les variables dans un programme et dans un goal sont considérées comme étant quantifiées universellement et existentiellement respectivement) alors que le comportement de Prolog sera indéfini sur ce goal. En effet, prouver ce goal revient à prouver le sous-goal `X ≤ Y` et ce built-in n'est défini que lorsque les deux paramètres sont ground (complétement instanciés) et sont de type entier.

- **unsoundness**

Prolog peut prouver certains goals qui ne sont pas des conséquences logiques du programme. Cela est dû principalement à la négation. L'absence d'occur-check peut également conduire Prolog à de mauvaises déductions.

Exemple 2.5. *(négation)*

```
p(X) ← X=a,not(p(b)).
```

Prolog va prouver que `p(a)` est vrai alors que ce n'est pas une conséquence logique du programme. En fait, c'est une conséquence logique de la *complétion* du programme, c-à-d `p(X)` ⇔ `X=a` ∧ `not(p(b))`. [1]

Exemple 2.6. *(négation)*

```
q ← not(p(b)).
p(Z) ← !,x=a.
p(b).
```

`q` peut être prouvé vrai alors que ce n'est une conséquence logique ni du programme ni de sa complétion.

Exemple 2.7. *(négation)*

```
q ← not(p(X)).
p(a).
```

---

1. [1],section 1.2.5.,p.9.

`not (q)` peut être prouvé vrai alors que ce n'est une conséquence logique ni du programme ni de sa complétion. Cela est dû au fait que $\exists$`Xp (X)` réussit (correctement) et donc Prolog en conclut (négation par échec) que $\exists$`Xnot (p (X) )` est faux, c-à-d que $\forall$`Xp (X)`. Une utilisation saine de la négation par échec demande donc que un `not` goal soit ground au moment de sa résolution.

Exemple 2.8. (occur-check)

```
p(X,X).
```

En l'absence d'occur-check, le goal `p (Y, f (Y) )` sera prouvé vrai alors que ce n'est pas une conséquence logique.

Ces exemples ont tenté de montrer qu'un programme Prolog ne doit pas être considéré simplement comme une théorie logique à partir de laquelle on peut déduire ses conséquences logiques. D'autres aspects plus procéduraux doivent être pris en compte tels que l'ordre des clauses, l'ordre des littéraux d'une clause, l'usage de built-ins extra-logiques et de la négation, le mode et le type des paramètres d'un goal au moment de sa résolution,... .
Une programmation Prolog sérieuse va donc demander une approche rigoureuse. La méthodologie de développement de procédures Prolog développée par Yves Deville est une première étape dans ce sens. Cette méthodologie est inspirée de l'équation de Kowalski : Algorithm = Logic + Control. [5]
La méthodologie est donc basée sur la logique et a pour idée de base de séparer les aspects logiques des aspects non logiques pendant le processus de construction d'une procédure Prolog: on résout le problème en ne s'attaquant d'abord qu'à ses aspects logiques, et ensuite on dérive de cette première solution la procédure finale en tenant compte des aspects extra-logiques. La méthodologie comprend trois étapes:

1. Elaboration d'une spécification: cette étape consiste essentiellement en la description de la relation à implémenter par la procédure ainsi que son domaine; des informations sur les conditions d'utilisation de la procédure sont également présentes.
2. Construction d'une description logique correcte par rapport à la spécification: formalisation en logique de premier ordre de la relation et de son domaine.
3. Dérivation d'une procédure Prolog correcte (et efficace) à partir de la description logique . C'est seulement lors de cette étape que seront traités les aspects non logiques propres à Prolog.

2.2. ELABORATION D'UNE SPECIFICATION.[1]

La spécification a pour but de décrire la relation (et son domaine) à implémenter par la procédure Prolog. Elle donne également des informations sur la façon dont sera utilisée la procédure. Ces informations ont la forme de pré-post conditions sur les paramètres de la procédure. D'autres renseignements tels que des pré-post conditions sur un environnement global peuvent être également précisés si nécessaire.
Nous présentons d'abord la forme générale d'une spécification. Le rôle de ses différents composants est expliqué par la suite.

---

1. [1],chapitre 2,p.25.

2.2.1. FORME GENERALE D'UNE SPECIFICATION.[1]

Pour l'élaboration de la spécification, aucun langage de spécification n'est imposé mais une forme précise et standard a été définie et se présente comme suit :

procédure `p(T1,...,Tn)`

Type : `T1: type1`
`...`
`Tn: typen`

Restriction sur les paramètres: description d'une relation *r* entre les paramètres `T1,...,Tn`

Relation: description d'une relation *p* entre les paramètres `T1,...,Tn`

Conditions d'application: *`Dir`*

Environnement global : pré-post conditions éventuelles sur un environnement global.

où
- `p` est le nom de la procédure Prolog;
- `T1,...,Tn` sont les paramètres formels de la procédure `p`;
- `type1,...,typen` sont les noms des *types* des paramètres formels. Un type représente un ensemble non vide de termes ground. Ainsi, le type *entier* pourrait représenter l'ensemble des entiers positifs, le type *boolean* représenterait `{true,false}` et le type *list* représenterait l'ensemble des listes, c-à-d les termes de la forme `[t1,...,tn]`.
  Remarquons que la relation existante entre le nom d'un type et l'ensemble qu'il représente n'est pas précisée. En fait, elle n'existe peut-être que dans la "tête" du programmeur comme elle pourrait être définie formellement au moyen d'un langage de description de type. Ce problème n'a pas d'importance ici (il le sera lorsque l'on tentera d'automatiser la méthodologie) et nous ne ferons donc pas de distinction entre le nom d'un type et l'ensemble qu'il représente.
- les relations *r* et *p* définissent chacune un ensemble de n-tuples ground terms.
- *`Dir`* est un ensemble de *directionnalités*. Une directionnalité décrit une manière d'utiliser la procédure. Elle décrit le mode des paramètres réels avant et après l'exécution de la procédure ainsi que le nombre de réponses calculées (multiplicité).
- L'environnement global d'une procédure désigne les "objects" accessibles par la procédure et qui peuvent exister en l'absence de son exécution. Les pré-conditions d'environnement définissent l'état que l'environnement doit satisfaire avant l'exécution de la procédure. Il est également nécessaire de spécifier les effets de bord (post-conditions) décrivant les effets que l'exécution de la procédure peut avoir sur l'environnement global (ex: avancement de la tête de lecture d'un fichier). Ce genre d'information nécessite un traitement tout particulier sortant du cadre de ce mémoire; nous ignorerons donc, à partir de maintenant, ce type de pré-post conditions.

Définition 2.1. (forme d'une directionnalité)

Une directionnalité d'une procédure `p/n` a la forme

1. [1],section 2.2.,p.29.

in (m1,...,mn) : :**out** (M1,...,Mn) <p-q>

où - mi,Mi $\subseteq$ {ground,var,ngv}
- mi,Mi $\neq \emptyset$
- p $\in$ N $\cup$ {∞}, q $\in$ N $\cup$ {*,∞}

Si d dénote une directionnalité, alors nous utiliserons les notations suivantes:
- d.in dénotera (m1,...,mn)
- d.out dénotera (M1,...,Mn)
- d.min dénotera p
- d.max dénotera q

Intuitivement, d.in (d.out) décrit le mode des paramètres avant(après) l'exécution de la procédure tandis que d.min (d.max) décrit le nombre minimum(maximum) de réponses calculées par la procédure(multiplicité). Ces notions sont expliquées plus formellement dans la section suivante.

Exemple 2.9. (directionnalité)

Quelques directionnalités possibles pour la procédure p/3:

in (ground,ground,ground) : :**out** (ground,ground,ground) <0-1>
in (ground,ground,any) : :**out** (ground,ground,ground) <1-1>
in (any,any,ground) : :**out** (ground,ground,ground) <1-*>
in (any,ground,any) : :**out** (novar,ground,novar) <1-∞>

Dans cet exemple et par la suite, nous utilisons les conventions suivantes: ground,var,ngv,novar, noground,gv et any représentent respectivement les ensembles {ground},{var},{ngv}, {ground,ngv} ,{var,ngv},{ground,var} et {ground,var,ngv}.

Exemple 2.10. (spécification de flattree/2)

procédure flattree(T,FT)

Type : T: *tree*
FT: *liste*

Restriction sur les paramètres: aucune.

Relation: { (t,ft) : t est la liste comprenant tous les éléments de ft dans l'ordre préfixé}

Conditions d'application:
in (ground,any) : :**out** (ground,ground) <0-1>
in (any,ground) : :**out** (ground,ground) <0-*>

Environnement global : aucune pré-post condition

2.2.2. ROLE DES DIFFERENTS COMPOSANTS.

L'ensemble des composants d'une spécification (`type1, ..., typen`,*p,r,Dir*) précise les préconditions et les postconditions de la procédure. Les préconditions définissent les appels corrects de la procédure, c-à-d les appels pour lesquels l'exécution est définie, tandis que les postconditions précisent l'état des paramètres après de tels appels.Si les paramètres ne respectent pas les préconditions, par convention, l'effet de la procédure est indéfini. Voyons, à présent, de manière plus précise ces deux notions de précondition et de postcondition ainsi que leur utilité.

- **PRE-CONDITION : Types + Restrictions sur les paramètres + partie "in" des directionnalités.**

Nous allons définir l'ensemble *Pre* représentant l'ensemble des termes respectant les préconditions de p/n. Cet ensemble est déterminé par les types (`type1, ... ,typen`), les restrictions sur les paramètres (*r*) et les parties "in" de l'ensemble des directionnalités (*Dir*).Définissons auparavant quelques concepts auxiliaires.

Définition 2.2. (Domaine de la relation: Dom)

> Soient `T1,T2,...,Tn` des termes ground.
> (`T1,T2,...,Tn`) $\in$ *Dom* ssi
> - (`T1,T2,...,Tn`) $\in$ (`type1 x type2 x ... x typen`) $\cap r$.

Définition 2.3. (Respect de mode)

> Le terme `X` respecte le mode `ground`, `var` ou `ngv` ssi `X` est respectivement un terme ground, une variable ou ni l'un ni l'autre.
> Le terme `X` respecte un ensemble de modes ($\subseteq$ {`ground, var,ngv`}) ssi `X` respecte au moins un des modes de cet ensemble. Ainsi `X` respecte {`ground, var`} ssi `X` est soit un terme ground soit une variable.
> Le terme (`X1,X2,...,Xn`) respecte (`m1,m2,...,mn`) ssi `Xi` respecte `mi`, $\forall$`i` (avec `m1,m2,...,mn` $\subseteq$ {`ground, var,ngv`}).

Pour que la pré-condition de la procédure soit respectée, non seulement les paramètres doivent appartenir au domaine de la relation (c-à-d avoir au moins une ground instance dans ce domaine) mais en plus, ils doivent satisfaire la partie "in" d'au moins une directionnalité spécifiée.

Définition 2.4. (PRE-CONDITION: Pre)

> Nous dirons que `X` = (`X1,X2, ... , Xn`) $\in$ *Pre* ssi
> - $X^* \cap Dom \neq \emptyset$
> - $\exists$`d` $\in$ *Dir* t.q. `X` respecte `d.in`

Rem: la notation X^* où `X` est un terme Prolog dénote l'ensemble {`t` | `t` est une ground instance de `X`}.

- **POST-CONDITION : Relation + Types + Restrictions sur les paramètres + partie "in/out" des directionnalités + multiplicité.**

La définition de la relation est la partie centrale de la spécification. Elle précise la relation à implémenter et peut être vue comme une post-condition car elle donne l'information sur les paramètres après une exécution réussie de la procédure. Intuitivement, la procédure sera correcte par rapport à la spécification

lorsque tout ce que l'on peut "calculer avec" est exactement la relation *p*. La notion de correction sera présentée de manière plus formelle dans la section 2.4.
Les types, les restrictions sur les paramètres et les directionnalités agissent aussi comme post-condition car ils donnent de l'information sur les paramètres après une exécution réussie de la procédure `p`.

Si la procédure `p/n` est appelée avec `X` $\in$ *Pre*, alors on peut dire que :
- n'importe quelle "substitution calculée" (c.a.s.) de `p(X)` devra respecter la partie "out" de toute directionnalité pour laquelle la partie "in" était respectée avant l'appel. De plus, cette substitution devra être compatible avec la relation et son domaine.
- le nombre de "c.a.s" de `p(X)` devra respecter la multiplicité de toute directionnalité pour laquelle la partie "in" était respectée avant l'appel.

Plus formellement, nous écrirons :

Définition 2.5. (POST-CONDITION)

Les postconditions seront respectées ssi
$\forall$ `X` $\in$ *Pre*, $\forall$ `d` $\in$ *Dir* | `X` respecte `d.in`, nous avons:
- $\forall \theta \in$ "c.a.s" de `p(X)`,
  - `X`θ respecte `d.out`,
  - $X\theta^* \cap Dom \neq \emptyset$
  - $X\theta^* \cap Dom \subseteq p$
- le nombre de c.a.s. de `p(X)` $\geq$ `d.min` et $\leq$ `d.max`

Remarque :
- Si `d.min` est un nombre fini et `d.max` est * alors le nombre de c.a.s. devra être un nombre fini $\geq$ `d.min`.
- Si `d.min` est un nombre fini et `d.max` est ∞ alors le nombre de c.a.s. devra être $\geq$ `d.min` et pourra être ∞.
- Si `d.min` est ∞ alors le nombre de c.a.s. ne pourra être que ∞.

- **Utilité des pré- et post-conditions.**

Pré-conditions et post-conditions sont particulièrement utiles à l'implémenteur et à l'utilisateur de la procédure puisqu'elles permettent à l'implémenteur de connaître les conditions qui doivent être respectées pour que l'appel de la procédure soit correct et à l'utilisateur de savoir quels seront les résultats d'un appel correct de la procédure logique.

2.3. CONSTRUCTION D'UNE DESCRIPTION LOGIQUE.

La description logique est élaborée à partir de la spécification. C'est une étape particulièrement créative basée sur la sémantique déclarative. La construction d'une procédure logique est totalement indépendante de Prolog et de sa sémantique procédurale. Seules les parties logiques de la spécification (la relation et son domaine) seront utilisées. Les directionnalités ne seront pas traitées ici.
La logique est souvent définie comme l'étude du raisonnement et est utilisée pour formaliser et symboliser le raisonnement. Ici,la logique de 1er ordre sera utilisée pour formaliser les descriptions logiques.
Nous allons définir ci-après la forme générale d'une description logique ainsi que la correction de la description logique par rapport à la spécification

2.3.1. FORME GENERALE D'UNE DESCRIPTION LOGIQUE. [1]

La description logique d'une procédure `p/n`, notée `DL(p/n)`, doit être une formalisation correcte de la relation et est une formule logique de 1er ordre de la forme :

p(X) ⇔ Def

où
- `X` est un n_tuple `(X1,X2, ... , Xn)` où `X1,X2, ... , Xn` sont les paramètres formels de la procédure et sont universellement quantifiés;
- `n ≥ 0` ;
- `Def` est une formule logique de 1er ordre où toute variable libre et distincte de `X1,X2, ... , Xn` est supposée quantifiée existentiellement en tête de `Def`.

Exemple 2.11. (Description logique de flattree/2)

```
flattree (T,FT) ⇔ T=void ∧ FT=[ ] ∨
                  T=tree(E,LT,RT) ∧ flattree(LT,FLT) ∧
                  flattree(RT,FRT) ∧ append(FLT,FRT,Tail_FT) ∧
                  FT=[E|Tail_FT].
```

2.3.2. CORRECTION DE LA DESCRIPTION LOGIQUE PAR RAPPORT A LA SPECIFICATION.[2]

Nous dirons qu'une description logique est correcte s'il existe une "équivalence" entre la relation décrite dans la spécification et les conséquences logiques de la description logique.
Nous allons définir plus formellement cette notion mais auparavant, nous définirons la correction d'un ensemble de descriptions logiques par rapport à une relation. Cette définition auxiliaire est nécessaire car une description logique n'a de sens qu'en tenant compte de ses sous-relations.
Nous dirons donc qu'un ensemble `SDL` de descriptions logiques décrit correctement la relation `p/n` ssi pour tout n_tuple `t`,

(C1) $SDL \models_H p(t)$ ssi $t \in p$ et $t \in Dom$

(C2) $SDL \models_H \sim p(t)$ ssi $t \notin p$ ou $t \notin Dom$

Enfin, nous dirons que `DL(p/n)` est correcte ssi ∀ `SDL` contenant `DL(p/n)` et décrivant corectement ses sous-relations, `SDL` décrit correctement `p/n`.
Ainsi, la description logique `DL(flattree/2)` présentée ci-dessus est correcte car n'importe quel `SDL` contenant `DL(flattree/2)` et décrivant correctement `=/2` (égalité) et `append/3` (concaténation de listes) décrira correctement la relation `flattree/2`.

N.B. : nous nous limitons ici à l'univers de Herbrand - l'ensemble de tous les termes ground - et aux

1. [1],section 3.4.,p69.
2. [1],section 4.1.,p74.

interprétations de Herbrand.Cette restriction d'un point de vue interprétation est une extension d'un point de vue conséquences logiques.L'ensemble des conséquences logiques de Herbrand est un super-ensemble de l'ensemble des conséquences logiques.

2.4. DERIVATION D'UNE PROCEDURE PROLOG CORRECTE.

Cette étape est l'étape finale de la méthodologie. Elle consiste en la dérivation d'une procédure Prolog correcte à partir d'une description logique. Cette procédure Prolog doit être correcte par rapport à la spécification.
Cette étape se base non plus sur la logique de 1er ordre mais sur la sémantique procédurale de Prolog. De plus, tous les aspects de la spécification sont pris en compte c-à-d relation, types, modes, environnement global et effets de bord.
Nous allons définir ci-après la forme générale d'une procédure Prolog ainsi que la correction de la procédure Prolog par rapport à la spécification

2.4.1. FORME GENERALE.[1]

Une procédure Prolog pour `p/n`, notée `PP (p/n)` est une séquence de clauses Prolog de la forme "`p(X) ← L1,L2,...,Lm.`"

où
- les `Li` sont des litéraux positifs ou négatifs ($m \geq 0$)
- `X` est un n_tuple `(X1,X2, ... , Xn)` où `X1,X2, ... , Xn` sont les paramètres formels de la procédure.
- les clauses d'une même procédure ont la même tête.

Exemple 2.12. (Procédure flattree/2)

```
flattree (T,FT) ← T=void , FT=[ ].
flattree (T,FT) ← T=tree(E,LT,RT) , flattree(LT,FLT) ,
                  flattree(RT,FRT) , append(FLT,FRT,Tail_FT) ,
                  FT=[E|Tail_FT].
```

2.4.2. CORRECTION DE LA PROCEDURE PROLOG PAR RAPPORT A LA SPECIFICATION.[2]

Informellement, nous dirons qu'une procédure Prolog est correcte s'il existe une "équivalence" entre la relation décrite dans la spécification et ce qui est calculé par la procédure, pour des appels réalisés avec des paramètres réels compatibles avec le domaine de la relation et satisfaisant les pré-conditions de directionnalité de la spécification.
De manière analogue à la définition de la correction d'une description logique par rapport à la spécification, nous définirons la correction d'une procédure Prolog et la correction d'une procédure Prolog dans un

1. [1],section 7.1.,p.187.
2. [1],section 8.2.,p.213.

ensemble de procédures Prolog (ou programme Prolog).
La correction d'une procédure Prolog sera indépendante des procédures Prolog pour ses sous-problèmes. La construction d'une procédure Prolog correcte reposera seulement sur la spécification des sous-problèmes et ne demandera pas la construction explicite des procédures Prolog correctes pour ses sous-problèmes.

Nous dirons que la procédure Prolog `PP (p/n)` est correcte ssi $\forall$ ensemble de procédures Prolog `SPP`, avec `SPP` $\supseteq$ `PP (p/n)` et `SPP` étant correct par rapport aux spécifications des sous-procédures de `PP (p/n)`, `SPP` est correct par rapport à la spécification de `p/n`.

Le concept de correction d'un `SPP` par rapport à la spécification d'une procédure `p/n` sera défini par un ensemble de contraintes entre ce qui est calculé par le `SPP` pour des appels corrects de `p/n` et ce qui est défini par la relation. Ces critères, présentés plus complétement ci-après, sont la correction partielle, la complétude, le respect des post-conditions, la non-redondance et la terminaison.

Nous dirons qu'un ensemble de procédures Prolog `SPP` est correct par rapport à la spécification d'une procédure `p/n` ssi $\forall$ `X` = (`X1`, `X2`, ... , `Xn`) $\in$ *Pre*,
la séquence de "c.a.s." `SUBST` calculée avec `SPP` $\cup$ {$\leftarrow$`p (X)`} respecte les conditions suivantes:

- **correction partielle.**

  $\{X\theta^* \cap Dom: \theta \in$ `SUBST`$\} \subseteq (Dom \cap p \cap X^*)$.

  La correction partielle exige que toutes les "bonnes" instances des réponses de `SUBST` apartiennent à la relation.

- **complétion.**

  $(Dom \cap p \cap X^*) \subseteq \{X\theta^* \cap Dom: \theta \in$ `SUBST`$\}$.

  La complétion exige que chaque élément de la relation puisse être calculé.

- **respect des post-conditions de la spécification.**

  $\forall \theta \in$ `SUBST`,
  - $X\theta^* \cap Dom \neq \emptyset$
  - $X\theta$ respecte `d.out`, $\forall$ `d` $\in$ *Dir* | `X` respecte `d.in`
  - `d.min` $\leq$ #`SUBST` $\leq$ `d.max`, $\forall$ `d` $\in$ *Dir* | `X` respecte `d.in`.

- **non-redondance.**

  $\forall \theta 1, \theta 2 \in$ `SUBST`, $(X\theta 1^* \cap Dom) \cap (X\theta 1^* \cap Dom) = \emptyset$.

- **terminaison.**

  S'il existe pour `p (X)` une séquence finie de "c.a.s." partiellement correcte et complète dans le sens défini ci-dessus, alors `SUBST` doit être fini.

  Si `SUBST` est fini, alors l'exécution de `SPP` $\cup$ {$\leftarrow$`p (X)`} se termine.

3. DERIVATION D'UNE PROCEDURE PROLOG CORRECTE A PARTIR D'UNE DESCRIPTION LOGIQUE CORRECTE.

La dérivation d'une procédure Prolog correcte est faite en deux étapes.La première consiste en une dérivation syntaxique à partir de la description logique et la deuxième en la vérification de certains critères de correction.
Pour obtenir une procédure Prolog correcte, la description logique correcte est d'abord transformée syntaxiquement en une procédure Prolog (avec négation). Intuitivement, cela revient à une sorte de "decompletion" puisque la procédure obtenue sera telle que sa "completion" Clark [4] est équivalente à la description logique. Notons qu'à ce stade, l'ordre des clauses n'a pas d'importance et une procédure est vue comme un ensemble de clauses au lieu d'une séquence de clauses.
Deville [1] a prouvé que la correction partielle et la complétude (du moins théorique) étaient assurées pour cette procédure, en s' appuyant sur un certain nombre d'hypothèses. La procédure sera donc correcte si ces hypothèses sont bien respectées, et si le reste des critères de correction (autre que la correction partielle et la complétion) définis dans le chapitre précédant le sont également. Ces hypothèses et ces critères forment ainsi un ensemble de nouveaux critères de correction, plus restrictif que celui présenté dans le chapitre précédent (section 2.4.2.) - le nouvel ensemble implique l'ancien mais pas l'inverse - mais aussi plus facilement vérifiable, dans une optique d'automatisation. L'objectif de ce chapitre est de rappeler rapidement ces nouveaux critères, ainsi que la phase de dérivation syntaxique.

3.1. DERIVATION SYNTAXIQUE D'UNE PROCEDURE PROLOG A PARTIR D'UNE DESCRIPTION LOGIQUE.[1]

Dans cette section,nous allons décrire rapidement la phase de transformations syntaxiques, accompagnée d'un exemple. Cette phase peut être complétement automatisée en utilisant les règles formelles de transformations décrites de facon complète dans Deville [1].

- **Les règles de dérivation.**

  1. Transformation initiale.

     La description logique `p(X)` $\Leftrightarrow$ *Def* est remplacée par `p(X)` $\Leftarrow$ *Def.*

  2. Transformations intermédiaires.

     Ces transformations ont pour but de transformer `p(X)` $\Leftarrow$ *Def* en `p(X)` $\Leftarrow$ *Def'*, avec *Def'* étant une disjonction de conjonction de littéraux équivalente à *Def.* Un ensemble de règles formelles de transformation a été defini dans ce but Deville [1].

Exemple 3.13. (Règle 1)

---

1. [1],chapitre 9,p.225.

Remplacer `p(X)⇐ c1∨...∨ L1∧...∧(F1∨F2)∧...∧Ln ∨...∨cm`
Par `p(X)⇐ c1∨...∨ L1∧...∧F1∧...∧Ln ∨ L1∧...∧F2∧...∧Ln ∨...∨cm`

3. Transformation finale.

`p(X)` ⇐ *Def'* est transformé en une procédure Prolog, de la manière suivante: chaque conjonction `L1∧...∧Li∧...∧Ln` produira la clause `p(X)` ←`L1*, ..., Li*, ..., Ln*`, où `Li*` est `q` et `not(q)`, selon que `Li` est respectivement `q` et `~q`.

Exemple 3.14. (Dérivation syntaxique de flattree/2)

- Description logique de flattree/2

```
flattree (T,FT) ⇔ T=void ∧ FT=[ ] ∨
                  T=tree(E,LT,RT) ∧ flattree(LT,FLT) ∧
                  flattree(RT,FRT) ∧ append(FLT,FRT,Tail_FT) ∧
                  FT=[E|Tail_FT].
```

- Transformation initiale.

```
flattree (T,FT) ⇐ T=void ∧ FT=[ ] ∨
                  T=tree(E,LT,RT) ∧ flattree(LT,FLT) ∧
                  flattree(RT,FRT) ∧ append(FLT,FRT,Tail_FT) ∧
                  FT=[E|Tail_FT].
```

- Transformations intermédiaires.

```
flattree (T,FT) ⇐ T=void ∧ FT=[ ] ∨
                  T=tree(E,LT,RT) ∧ flattree(LT,FLT) ∧
                  flattree(RT,FRT) ∧ append(FLT,FRT,Tail_FT) ∧
                  FT=[E|Tail_FT].
```

- Transformation finale.

```
flattree (T,FT) ← T=void , FT=[ ].
flattree (T,FT) ← T=tree(E,LT,RT) , flattree(LT,FLT) ,
                  flattree(RT,FRT) , append(FLT,FRT,Tail_FT) ,
                  FT=[E|Tail_FT].
```

Remarque : La dérivation syntaxique à partir d'une seule description logique peut conduire à plus d'une procédure logique.Ce cas est présenté dans [Deville] et fait suite à une particularité d'une des règles de dérivation.

3.2. NOUVEAUX CRITERES DE CORRECTION.[1]

1. [1],section 9.3.,p.230.

Il a été prouvé que si PP (p/n) est dérivée syntaxiquement d'une description logique correcte DL (p/n), alors PP (p/n) est correct par rapport à sa spécification (section 2.4.2) si dans toute exécution SPP ∪ {←p (X) }, avec

- SPP étant n'importe quel ensemble de procédures Prolog tel que SPP ⊇ PP (p/n) et SPP étant correct par rapport aux spécifications des sous-procédures de PP (p/n)
- X = (X1, X2, ... , Xn) ∈ *Pre*,
- SUBST dénommera la séquence de "c.a.s." calculées avec SPP ∪ {←p (X) }[1],

les conditions suivantes sont respectées:

a) **appel correct des sous-procédures.**

∀q(Y) sélectionné dans PP (p/n), Y doit être compatible avec le domaine de q, c-a-d $Y^* \cap$ *Dom* (q) ≠ ∅. Si le littéral sélectionné est positif, alors ∃d ∈ *Dir* (q) | Y respecte d.in sinon les arguments doivent être ground.

b) **préservation du domaine("domain consistency").**

Nous savons que les arguments d'un littéral de PP (p/n) juste après son exécution sont compatibles avec le domaine de ce littéral, puisque l'on suppose que SPP est correct par rapport aux spécifications des sous-procédures de PP (p/n). Par contre rien ne dit que cette compatibilité persistera lors de la résolution des littéraux suivants. Ce caractère extra-logique peut nuire à la correction partielle ou à la complétion, ou les deux. On assurera donc cette persistance par la condition suivante:

∀ clause p (Y) ← L1 (t1), ..., Lm (tn)
∀ "c.a.s" θ de SPP ∪ {←Y=X, L1 (t1), ..., Lm (tn) }
∀ s ∈ $X\theta^*$ ∩ *Dom* (p), ∃ s1, ..., sm:
(s, s1, s2, ..., sm) ∈ (X, t1, ..., tm) θ^* ∩ (*Dom* (P) Ξ *Dom* (L1) Ξ ... X *Dom* (Lm))

c) **complétude pratique.**

∀ "c.a.s" *théorique*[2] θ de SPP ∪ {←p (X) }, θ ∈ SUBST.

d) **respect des postconditions sur les modes et les types.**

∀ θ ∈ SUBST,
- $X\theta^*$ ∩ *Dom* ≠ ∅
- Xθ respecte d.out, ∀ d ∈ *Dir* | X respecte d.in.

e) **respect des postconditions sur les multipicités.**

d.min ≤ #SUBST ≤ d.max, ∀ d ∈ *Dir* | X respecte d.in.

f) **non-redondance.**

∀ θ1, θ2 ∈ SUBST, ($X\theta1^*$ ∩ *Dom*) ∩ ($X\theta1^*$ ∩ *Dom*) = ∅.

g) **terminaison.**

---

1. Ici, l'ordre des clauses dans SPP est bien sûr important.
2. Par c.a.s. théorique nous entendons toute c.a.s. existant dans l'arbre de SLD-derivation, peu importe qu'elle soit effectivement atteinte ou non par la règle de dérivation PROLOG.

S'il existe pour `p(X)` une séquence finie de "c.a.s." partiellement correcte et complète dans le sens défini en 2.4.2., alors `SUBST` doit être fini.

Si SUBST est fini, alors l'exécution de SPP $\cup$ {$\leftarrow$p(X)} se termine.

h) occur-check.

Le prédicat `=/2` est supposé être implémenté correctement; cela signifie qu'un goal de la forme `=(X,f(X))` doit échouer.

L'analyseur que nous nous proposons d'étudier dans ce mémoire s'occupe des critères **a)** et **d)**, c-à-d des critères sur les modes et les types. Notons que le critère **b)** est lui aussi lié aux types et aux modes mais n'est pas traité actuellement.

4. DESCRIPTION ET PRESENTATION DE L'ANALYSEUR.

4.1. INTRODUCTION.

Cet analyseur a pour but de dériver une procédure Prolog correcte à partir de sa description logique.Pour l'instant, la vérification des critères se limite aux types et aux modes des paramètres c-à-d les critères **a)** et **d)** présentés dans le chapitre précédent (section 3.2.).
Cela revient à s'assurer que pour chaque clause de la procédure, pour chaque appel correct, l'exécution de la clause est telle que :

- chaque sous-procédure est appelée correctement
- les post-conditions sont respectées en fin d'exécution.

Un analyseur statique pouvant calculer *at compile-time* des informations sur le mode et le type des variables d'une clause lors de son exécution constituera donc le noyau central de l'analyseur chargé de dériver des procédures correctes.
Ce noyau est spécifié dans la section 4.3, précédée d'abord d'une rapide présentation du concept d'analyse statique de clauses Prolog. L'analyseur, ainsi que deux autres outils auxiliaires, eux aussi construits à partir de ce noyau sont présentés dans la section 4.4.

4.2. ANALYSE STATIQUE DE CLAUSES PROLOG.

L'analyse statique d'une clause Prolog consiste à déduire des informations sur l'exécution de cette clause, lorsque celle-ci est exécutée avec n'importe quel goal respectant un certain ensemble de propriétés. Cet ensemble de propriétés est désigné sous le nom de goal *abstrait*.
Ce type d'analyse est dit statique car l'analyse se fait non pas par une exécution réelle de la clause mais par une *interprétation abstraite* de son texte source. Ce type d'interprétation mime une exécution réelle mais opère sur des ensembles de substitutions (pouvant être infinis) au lieu de substitutions. Un ensemble de substitutions est réprésenté par une *substitution abstraite*, dont la syntaxe est évidemment propre à chaque système mais dont la sémantique est habituellement exprimée au moyen d'une fonction de *concrétisation* qui définit pour chaque substitution abstraite l'ensemble des substitutions concrètes qu'elle représente.
L'analyse statique d'une clause Prolog revient en fait à calculer, pour une substitution abstraite de départ (le goal abstrait) une substitution abstraite à chaque point d'exécution de la clause, c-à-d devant chacun de ses littéraux et à la fin de la clause.

La qualité d'une analyse statique peut être jugée selon les quatre critères principaux suivants:

correction: ce critère est essentiel pour utiliser correctement les informations déduites. Il impose que l' interprétation abstraite effectuée soit une "bonne" approximation de toutes les exécutions réelles possibles, c-à-d que pour n'importe quelle substitution concrète θ pouvant "arriver" à n'importe quel point d'exécution `ptx` lors d'une exécution lancée avec un goal respectant le goal abstrait, il faut que θ soit réellement représentée par la substitution abstraite calculée en `ptx`. Ainsi, si une substitution abstraite en un point spécifie que `X` est `ground`, alors il faut qu'effectivement `X` soit `ground` dans n'importe quelle substitution concrète pouvant survenir en ce point.

terminaison: une interprétation abstraite n'est pas un but en soi mais vise à déduire des informations qui seront utilisées automatiquement par d'autres outils (compilateur, générateur de code, ...). La terminaison est donc impérative dans cette optique.

efficacité: nous entendons par là la rapidité avec laquelle les résultats sont calculés. Notons que ce critère sera plus déterminant dans certaines applications, comme l'optimisation de code produit par un compilateur.

précision: la substitution abstraite idéale en un point est celle (si elle existe) représentant exactement l'ensemble des substitutions arrivant réellement en ce point. Plus une substitution abstraite se "rapproche" de cet idéal, plus elle sera précise. Un des facteurs déterminant pour ce critère est l'expressivité du langage choisi pour définir les substitutions abstraites.
Notons que ce critère peut s'opposer au critère de terminaison et d'efficacité. Remarquons aussi que le degré de précision demandé variera d'une application à l'autre. Ainsi, un type d' application très fréquent consiste à déterminer si une variable est ground ou non. Dans ce type d'application, `X` est `ground` sera considéré comme la substitution abstraite idéale et toute précision supplémentaire (ex: `X` est `ground` et est une `liste`) est considérée comme superflue.

4.3. SPECIFICATION DE L'ANALYSEUR STATIQUE D'UNE CLAUSE (NOYAU).

L'Analyseur étudié dans ce mémoire utilise un analyseur statique calculant des substitutions abstraites exprimant des informations sur les *types* et les *modes* des variables de la clause, ainsi que des informations sur les liens entre ces variables, cette dernière information visant à améliorer la précision des résultats obtenus.
L'ensemble des types (et modes) reconnus par l'analyseur n'est pas fixé une fois pour toutes, par un formalisme précis du style grammaire `BNF`, ... En fait n'importe quel ensemble non vide de termes (incluant des termes non ground) peut être représenté par un type, le lien entre le nom du type et l'ensemble qu'il représente étant exprimé plus ou moins précisément par l'instantiation d'un ensemble de primitives Prolog. Ces primitives donnent des informations de base sur la sémantique des types et sont utilisées lors de l'analyse statique. Elles jouent donc le rôle d'un interface entre le noyau et les types, permettant ainsi l'implémentation d'un noyau générique, indépendant de tout système de types particulier. Remarquons au passage qu'un mode n'est en fait qu'un type particulier dans ce système, puisque le concept de type n'est pas limité qu'aux seuls termes grounds.
L'analyse statique se basera non seulement sur le texte de la clause mais utilisera aussi ce que l'on appelle les *behaviours* associés à ses sous-procédures. Le behaviour d'une procédure est en fait une spécification abstraite de celle-ci, en forme de pre-post conditions sur ses paramètres formels, où chaque pre/post condition sera réellement une substitution abstraite. Ces behaviours sont indispensables pour passer d'un point d'exécution au suivant.
Nous allons maintenant définir plus précisement ces différents concepts - *type*, *substitution abstraite*, *primitive* et *behaviour* - et nous terminerons par la spécification du noyau.

4.3.1. TYPE.

Un *type* est simplement un nom, sensé représenter un ensemble non vide de termes. Un *système de types* est un couple (TYPE,γ), avec

- TYPE étant un ensemble fini de types.
- γ est la fonction de concrétisation de ces types, c-à-d une fonction définie comme $\mathtt{TYPE} \rightarrow 2^{\mathtt{PROLOG}}$, ou PROLOG représente l'ensemble des termes.

Un système de type sera défini par l'utilisateur en fonction de ses besoins et la fonction γ sera formalisée au moyen des primitives de types décrites en 4.3.3. Tout système de type sera accepté, pour autant qu'il respecte les contraintes suivantes, nécessaires pour une analyse statique correcte et précise:

- TYPE contient au moins les types var, ground et any; ce sont les types primitifs.
- γ(var)={t: t est un terme variable}
- γ(ground)={t: t est un terme ground}
- γ(any)={t: t est un terme}
- $\forall$ T$\in$TYPE, $\forall$ t $\in$ γ(T): t' $\in$ γ(T) si t' est un renommage de t, c-à-d t'=t•<x1/y1,...,xn/yn> avec x1,...,xn étant les variables de t et y1,...,yn étant des variables distinctes.

Exemple 4.15. (système de types)

TYPE = {var,ground,any,integer,integer_list,groundlist,groundtree, anylist,anytree}

γ(integer)={t: t est un entier}
γ(integer_list)={t: t est une liste d'entiers}
γ(groundlist)={t: t est une ground list}
γ(groundtree)={t: t est un ground tree}
γ(anylist)={t: t est un terme pouvant être instancié à une ground list}
γ(anytree)={t: t est un terme pouvant être instancié à un ground tree}

4.3.2. SUBSTITUTIONS ABSTRAITES.

Une substitution abstraite représente un ensemble de substitutions concrètes sur le même domaine et consiste en une liaison de variables (le domaine) à des termes abstraits, appelés *typed terms*.Ces termes sont des termes Prolog mais à la place de variables, ils utilisent des *types indexés*, qui sont des termes spéciaux de la forme $typ(i) où typ est un type et i un entier. De plus, la substitution abstraite contient un ensemble de contraintes "nosharing" exprimées sous la forme de couples de types indexés.
Les substitutions concrètes (supposées idempotentes) représentées par une substitution abstraite peuvent être exprimées intuitivement de la façon suivante: remplacer chaque type indexé dans la substitution abstraite par un terme Prolog respectant les contraintes suivantes :

(i) le terme doit être "consistant" avec le type du type indexé
(ii) deux mêmes types indexés (même type et même index) doivent recevoir les mêmes termes
(iii) deux types indexés doivent recevoir des termes sans variables communes si une contrainte "nosharing" existe entre eux.

Présentons maintenant de manière plus précise successivement les concepts de typed term et de substitution abstraite.

- **typed term**

syntaxe:

Un typed term est ou bien un type indexé `$typ(i)` ou bien un terme `f(tt1,...,ttn)` où `f/n` est un foncteur Prolog et `tt1,...,ttn` sont des typed terms.

sémantique:

L'ensemble des termes représentés par un typed term `tt`, dénommé γ`(tt)` est = {`t:t` est un terme Prolog obtenu à partir de `tt` en substituant à chacun de ses types indexés `$typ(i)` un terme Prolog appartenant à γ`(typ)`}.

Exemple 4.16. (typed term)

```
$var(1)
$anylist(1)
tree($any(1),$anytree(1),$anytree(2))
```

- **substitution abstraite**

syntaxe:

Une substitution abstraite est ou bien $\perp$ ou bien un couple `({x1/tt1,..., xn/ttn},nosh)` où :

- `x1,...,xn` `(n ≥ 0)` sont des variables distinctes et forment le domaine de la substitution
- `tti` est un "typed term"
- `nosh` est un ensemble de couples `($typa(i),$typb(j))` où `$typa(i),$typb(j)` doivent être des types indexés distincts utilisés dans `{tt1,...,ttn}`

sémantique:

L'ensemble des substitutions concrètes exprimées par une substitution abstraite sera exprimé par une autre fonction γ définie comme suit :

- $\gamma(\perp) = \varnothing$
- γ(`({x1/tt1,..., xn/ttn},nosh)`) = {`{x1/t1,..., xn/tn} : [t1,...,tn]` est un terme Prolog obtenu à partir de `[tt1,...,ttn ]` en substituant à chacun des types indexés $typ(i), un terme Prolog appartenant à γ`(typ)`.De plus, si `($typa(i),$typb(j))` $\in$ `nosh` alors le terme substitué à `$typa(i)` n'a pas de variables communes avec le terme substitué à `$tyb(j)`}.

Exemple 4.17. (substitution abstraite)

```
({L1/$groundlist(1),L2/$groundlist(2),L3/$anylist(1)},∅)
({X/$var(1),Y/$var(2)},{~($var(1),$var(2))})
({X/$ground(1),L/$groundlist(1),
L1/[$any(1),$anylist(1)]},{~($any(1),$anylist(1))})
```

4.3.3. PRIMITIVES.

Ces primitives visent à informer sur la fonction de concrétisation γ, pour une instantiation particulière (TYPE,γ).Il y a seulement cinq primitives.

a) ($\subseteq$_tterm_type/2)

Cette primitive reçoit un typed term `tt` et un type `typ` et renvoie la valeur 'vrai' seulement si γ(`tt`) $\subseteq$ γ(`typ`)

b) ($\subseteq$_type_ttermfct/2)

Cette primitive reçoit un type `typ` et un typed term `tt` de la forme `f(tt1,...,ttn)` et renvoie la valeur 'vrai' seulement si γ(`typ`) $\subseteq$ γ(`f(tt1,...,ttn)`)

c) (conj_tterm_type/2)

Cette primitive reçoit un typed term `tt` et un type `typ` et renvoie une substitution θ reliant tous les types indexés de `tt` à des typed terms tels que :

- `domaine` (θ) $\cap$ `codomaine` (θ) = $\varnothing$ (θ doit être idempotente)
- γ(`tt`θ) $\supseteq$ γ(`tt`) $\cap$ γ(`typ`)

Cette primitive peut aussi renvoyer $\perp$ ce qui signifie γ(`tt`) $\cap$ γ(`typ`) = $\varnothing$.
Une réponse triviale est la substitution reliant tous les types indexés de `tt` à un nouveau type indexé du même type.

d) (unif_tterm_type/2)

Cette primitive reçoit un typed term `tt` et un type `typ` et renvoie une substitution θ reliant tous les types indexés de `tt` à des typed terms tels que :

- `domaine` (θ) $\cap$ `codomaine` (θ) = $\varnothing$ (θ doit être idempotente)
- γ(`tt`θ) $\supseteq$ { `t` : $\exists$ `t1` $\in\gamma$(`tt`), $\exists$ `t2` $\in\gamma$(`typ`) tels que σ est un mgu de (`t1,t2`) et `t` = `t1`σ = `t2`σ }

Cette primitive peut aussi renvoyer $\perp$ ce qui signifie que l'ensemble ci-dessus est vide.
Une réponse triviale est la substitution reliant tous les types indexés de `tt` à des types indexés de type `any`.

e) (inst_type/1)

Cette primitive reçoit un type `typ` et renvoie un type `typinst` tel que γ(`typinst`) $\supseteq$ { `t` : $\exists$ `t1` $\in\gamma$(`typ`) tel que `t` est une instantiation de `t1` }
Une réponse triviale est le type `any`.

Remarque: L'instantiation des primitives n'est pas une tâche particulièrement aisée pour l'utilisateur.La version actuelle du système libère l'utilisateur de cette charge en générant automatiquement les primitives pour un ensemble de types donnés.Cette génération automatique n'est possible qu'au prix de deux concessions:

1. Expressivité: Les types doivent être décrits au moyen d'un formalisme bien défini ayant la forme d'un ensemble de clauses de Horn (avec certaines restrictions).Les types non exprimables dans ce formalisme devront être traités comme avant c-à-d instanciés manuellement.Les types sont en fait des types primitifs pour le système.

2. Précision: Les primitives générées ne seront pas toujours aussi précises que si elles avaient été instanciées manuellement.Le système permet néanmoins d'accéder directement au code généré et d'y faire les modifications éventuelles (inconvénient: risque d'inconsistances).

4.3.4. BEHAVIOURS

Nous devons analyser des clauses Prolog auxquelles est attaché un behaviour pour chacune de leurs sous-procédures.Un behaviour est un ensemble de pré- et post-conditions sur les modes et les types des paramètres formels de la procédure, avec en plus de l'information sur le 'nosharing'.Les substitutions abstraites expriment convenablement ces conditions.

Syntaxe:

Un behaviour pour la procédure `p/n` a la forme (`p/n, [y1,...,yn],prepost`) où :

- `y1,...,yn` sont des variables distinctes représentant les paramètres formels de la procédure
- `prepost` est un ensemble, peut-être vide de directionnalités `pre::post` où `pre` (`post`) est une substitution abstraite dont le domaine est {`y1,...,yn`} si elle est distincte de $\perp$.

Sémantique:

Intuitivement, un programme `P` est consistant avec le behaviour (`p/n, [y1,...,yn],prepost`) ssi les valeurs renvoyées après l'exécution de `p/n` dans `P` respectent la post-condition `post` si les valeurs avant l'appel respectent la pre-condition `pre`, pour chaque directionnalité `pre::post` appartenant à `prepost`.

Un programme Prolog `P` est consistant avec le behaviour (`p/n, [y1,...,yn],prepost`) ssi

- $\forall$ `pre::post` $\in$ `prepost`,
- $\forall$ {`y1/t1,...,yn/tn`} $\in \gamma$(`pre`),
- $\forall$ θ résultant de l'exécution de `P` $\cup$ {$\leftarrow$ `p(t1,...,tn)`},
  {`y1/t1`θ, ...,`yn/tn`θ} $\in \gamma$(`post`).

Exemple 4.18. (Behaviour)

```
flattree(T,FT)

T/$groundtree(1),FT/$anylist(1)::
T/$groundtree(1),FT/$groundlist(1).
```

```
T/$anytree(1),FT/$groundlist(1)::
T/$groundtree(1),FT/$groundlist(1).
```

4.3.5. SPECIFICATION DU NOYAU.

input:
- CL, une clause Prolog p(x1,...,xn) ← L1,...,Lf où n,f ≥0 et x1,...,xn sont des variables distinctes
- BEHAV, un ensemble de comportements; il contient un behaviour pour chaque sous-procédure q/s (distincte de =/2) utilisée dans L1,...,Lf
- asin, une substitution abstraite dont le domaine est x1,...,xn s'il est distinct de ⊥

output:
- as0,as1,...,asf qui sont des substitutions abstraites dont le domaine (si distinct de ⊥) est x1,...,xn,y1,...,ym c-à-d l'ensemble des variables apparaissant dans la clause CL
- asout, une substitution abstraite exprimant les mêmes propriétés que asf mais limitées à x1,...,xn

Ces substitutions abstraites calculées sont telles que :
- γ(as0) = s0, qui est l'ensemble { {x1/t1,..., xn/tn,y1/s1,...,ym/sm} : s1,...,sm sont des variables distinctes non utilisées dans t1,...,tn et {x1/t1,..., xn/tn} ∈γ(asin)}
- γ(asi) ⊇ {θσ, limité au domaine de θ : θ ∈ s0 et σ est une "computed answer substitution" de P ∪ {← (L1,...,Li)θ}, avec P étant tout programme consistant avec tous les behaviours de BEHAV} (∀ 1≤i≤f).

Le noyau renvoie un ensemble de substitutions abstraites intermédiaires et une finale, chacune exprimant les propriétés des substitutions concrètes apparaissant aux différents points de la clause pendant son exécution dans un programme P consitant avec tous les behaviours de BEHAV et commençant avec la substitution asin.

4.4. OUTILS DEVELOPPES A PARTIR DU NOYAU.

Trois outils ont été développés à partir de l'analyseur statique de clause. Le principal de ces outils est l'analyseur proprement dit, dénommé *SYNTHES* qui dérive une procédure Prolog correcte à partir de la description logique; deux autres outils, dénommés respectivement *DETAIL* et *PERMUT* ont été développés simultanément et ont pour objectif principal d'être utilisés comme supports auxiliaires de *SYNTHES*. Le reste de ce mémoire est consacré entièrement à l'utilisation de ces outils; nous nous limiterons donc ici à donner une brève description de chacun d'eux, consistant en deux parties: spécification et implémentation. La deuxième partie est ambitieusement nommée, et se limite en fait en une explication grossière de la manière dont l'outil utilise le noyau .

4.4.1. *DETAIL:* ANALYSE DETAILLEE D'UNE CLAUSE.

- **spécification**[1]

input: - CL, une clause Prolog p(x1,...,xn) ← L1,...,Lf où n,f ≥ 0 et x1,...,xn sont des variables distinctes
- pre::post,une directionnalité sur les paramètres de la tête de la clause CL et pre constitue asin.

output: - as0,as1,...,asf qui sont des substitutions abstraites dont le domaine (si distinct de ⊥) est x1,...,xn,y1,...,ym c-à-d l'ensemble des variables apparaissant dans la clause CL
- asout, une substitution abstraite exprimant les mêmes propriétés que asf mais limitées à x1,...,xn
- asout consistant ou non avec post.

- **implémentation**

DETAIL, après avoir contrôlé la correction de ses "inputs", va effectuer une analyse statique à partir de pre. Il calculera ensuite, pour chaque litéral Li de CL, la substitution abstraite juste avant son appel et s'assurera ensuite que cet appel est correct par rapport aux behaviours définis pour chacun de ces littéraux.Il s'assurera enfin que la dernière substitution abstraite calculée est compatible avec post.

4.4.2. *PERMUT:* PERMUTATIONS CORRECTES D'UNE CLAUSE.

- **spécification**

input: - CL, une clause Prolog p(x1,...,xn) ← L1,...,Lf où n,f ≥ 0 et x1,...,xn sont des variables distinctes
- pre::post,une directionnalité sur les paramètres de la tête de la clause CL et pre constitue asin.

output: - {CLi} avec i ≥ 0

- **implémentation**

PERMUT, après avoir contrôlé la correction de ses "inputs", calcule, pour autant que ce soit possible, une ou plusieurs permutations correctes des littéraux de la clause CL.En réalité, PERMUT procéde comme suit : il calcule, si possible, une première permutation correcte et demande ensuite à l'utilisateur s'il désire poursuivre ou non la recherche de permutations.

4.4.3. *SYNTHES:* DERIVATION D'UNE PROCEDURE PROLOG CORRECTE A PARTIR D'UNE DESCRIPTION LOGIQUE.

- **spécification**

input: - -DL(p/n), une description logique correcte

---

1. Pour chacun des outils, on suppose que chaque sous-procédure de CL est spécifiée par un behaviour.

output: - -PP (p/n), une procédure Prolog correcte dérivée de DL (p/n)
- -{PPi (p/n) } avec i ≥ 0

- **implémentation**

Le troisième et dernier outil développé, SYNTHES, reçoit en entrée une description logique correcte DL (p/n) .
Il transforme ensuite cette description logique en un ensemble de clauses PP (p/n), procédure Prolog.
Pour chaque directionnalité pre::post, du behaviour de la tête de DL (p/n) , pour chaque clause CL de PP (p/n), il essaye de calculer une permutation correcte de CL respectant pre::post.
A chaque permutation trouvée, l'utilisateur de SYNTHES peut arrêter le processus ou poursuivre la recherche d'autres permutations.
Une autre version de SYNTHES donnera, pour chaque CL de PP (p/n), les permutations correctes et communes à toutes les directionnalités.

Nous venons de spécifier l'Analyseur et d'étudier les outils développés pour sa réalisation. Que peut-on dire de son efficacité, des résultats obtenus? Sont-ils fiables, suffisamment précis?
C'est précisément à l'étude critique de ces résultats et du fonctionnement de l'analyseur que nous allons nous attacher dans le chapitre suivant.Nous étudierons l'analyseur à un niveau local (une clause) - DETAIL et PERMUT - puis global - SYNTHES -.
Nous tenterons enfin d'appliquer la méthodologie de Deville [1] et l'analyseur à une petite application concrète de Gestion de Commandes Clients et Fournisseurs.

5. UTILISATION DE L'ANALYSEUR DANS LA DERIVATION PROLOG.

5.1. INTRODUCTION.

Ce chapitre va être divisé en trois parties : l'étude de procédures classiques, l'étude d'une application simple de base de données et une conclusion générale. Dans la première, nous allons utiliser l'analyseur pour dériver et tester des procédures classiques. Ces procédures seront les suivantes : `efface/3`, `member/2`, `length/2`, `reverse/2` et `append/3`.
La deuxième partie de ce chapitre sera consacrée à l'étude d'une application simple de base de données. Dans un premier temps, nous présenterons le cas - une gestion de Commandes Clients et Fournisseurs - et dans un deuxième, nous définirons une query que nous analyserons selon la même découpe que les procédures classiques.
La troisième partie consistera à tirer des conclusions générales obtenues grâce à l'utilisation de l'analyseur.

5.2. UTILISATION LORS DE LA DERIVATION DE PROCEDURES CLASSIQUES.

Dans cette section, nous allons donc étudier les procédures classiques `efface/3`, `member/2`, `length/2`, `reverse/2` et `append/3`. Trois sous-sections seront consacrées à chacune d'entre elles. Dans la première, nous présenterons la spécification et la description logique de la procédure; dans la deuxième, nous expliquerons la façon dont les tests ont été réalisés ainsi que les résultats et dans la troisième sous-section, nous commenterons les résultats des différents tests et tâcherons d'en tirer des conclusions.
La procédure `efface/3` a été étudiée de façon plus approfondie par rapport aux autres procédures. Elle a permis de faire de premières remarques et conclusions. Celles-ci ont alors quelque peu orienté les tests des autres procédures; ceci afin de nous permettre de confirmer ou infirmer ces conclusions, que nous ne voulions pas trop hâtives.

5.2.1. CAS 1 : *efface/3*.

Des tests nombreux et variés ont été effectués sur cette procédure. Ceux que nous avons estimés plus significatifs et qui nous ont permis de tirer des conclusions sont présentés dans la deuxième partie de cette sous-section.

5.2.1.1. Spécification et description logique

La procédure `efface/3` est construite pour enlever, d'une liste, son premier élément égal à `X` et donner en résultat la liste `LEff`. `LEff` ne sera pas définie si `X` n'appartient pas à `L`. Nous essayerons que cette procédure soit multi-directionnelle autant que possible. Une procédure `PROLOG` est multi-directionnelle si elle fonctionne de plusieurs manières; les arguments pouvant être instanciés de façons diverses et donc être ou bien des données ou bien des résultats. Cependant, jamais les deux premiers arguments ni les deux

derniers de `efface/3` ne pourront être libres simultanément.

La spécification de `efface(X,L,LEff)` est donc la suivante :

procédure `efface (X,L,LEff)`

Type : `X: term`
`L,LEff: list`

Relation: `X` est un élément de `L` et `LEff` est la liste `L` sans la première occurrence de `X` dans `L`.

Conditions d'application:
in `(ground,ground,any)` **::out** `(ground,ground,ground)<0-1>`
in `(ground,any,ground)` **::out** `(ground,ground,ground)<0-n>`
in `(any,ground,any)` **::out** `(ground,ground,ground)<0-n>`

La description logique qui en est déduite prend la forme suivante :

```
efface (X,L,LEff) ⇔  L=[] ∧ false ∨
                     L=[H|T] ∧ (H=X ∧ LEff=T ∧ list(T) ∨ ~(H=X) ∧
                     efface(X,T,TEff) ∧ LEff=[H|TEff]).
```

5.2.1.2. Tests et résultats.

Lors de la troisième étape de la méthodologie, la desciption logique va d'abord être traduite syntaxiquement en deux clauses (sans le type checking `list/1`):

```
efface (X,L,LEff) ←  efface(X,T,TEff),=(LEff,[H|TEff]),not(=(H,X)),
                     =(L,[H|T]).
efface (X,L,LEff) ←  =(H,X),=(LEff,T),=(L,[H|T]).
```

L'analyseur *FOLON*, grâce aux outils *DETAIL*, *PERMUT* et *SYNTHES*, va nous permettre de trouver toutes les permutations correctes des clauses, qui respectent la ou les directionnalités. Notons au passage que chaque directionnalité est transmise à l'analyseur sous la forme d'un `pre::post` où `pre(post)` est la partie **in(out)** combinée avec les types. Le behaviour de la procédure correspond ainsi à l'ensemble de ces `pre::post` plus le behaviour explicite (s'il existe) de la procédure. Les résultats de ces analyses sont présentés ci-dessous tandis que les commentaires constituent une autre sous-section.
Avant de présenter le tableau des tests et résultats, nous pouvons remarquer deux choses. Il est possible de faire varier la spécification de `efface/3` (ou toute autre procédure) pour essayer de la rendre multi-directionnelle, pour trouver un nombre plus grand de permutations, pour trouver des permutations communes à plusieurs usages. Par exemple, il est possible de supprimer la contrainte de type sur l'un ou l'autre des arguments. Les directionnalités sont alors remplacées explicitement par des couples `pre::post` où `pre` et `post` sont des substitutions abstraites sur les paramètres de la procédure. Ces `pre::post` permettent d'exprimer des contraintes plus fines sur les paramètres. De cette façon, le domaine d'application de la procédure est élargi. Une autre façon d'agir est de dire que `any` est un mode

trop général et de le réduire à la combinaison `ground` et `var`. Le domaine d'application de la procédure est ainsi réduit. Il est également possible d'ajouter des contraintes explicites de *nosharing* entre les arguments.
Dans le tableau des résultats que nous présentons ci-dessous, nous trouverons en ordonnée, les usages testés - seulement leur partie **in** car la partie **out** est identique pour tous : `ground`, `groundlist`, `groundlist` - et en abscisses, dans les deux premières colonnes, les clauses de la procédure; la troisième colonne contiendra le nombre de permutations communes aux deux clauses. Chaque coordonnée des deux premières colonnes reprendra le nombre de permutations correctes pour la clause concernée et pour l'usage sélectionné.

Pour `efface/3`, les deux clauses testées sont numérotées comme suit:

- **Clause 1 :**

```
efface (X,L,LEff) ← efface(X,T,TEff),=(LEff,[H|TEff]),not(=(H,X)),
                    =(L,[H|T]).
```

- **Clause 2 :**

```
efface (X,L,LEff) ← =(H,X),=(LEff,T),=(L,[H|T]).
```

Les tests vont être résumés dans un tableau ci-après mais auparavant, il est utile d'apporter les précisions qui suivent. Dans les tableaux de ce chapitre, `g` signifie `ground`, `a` signifie `any`, `v` signifie `var`, `gl` signifie `groundlist`, `al` signifie `anylist`, `int_a` (`i_a`) signifie `integer_any`.Une `anylist` est un terme qui peut être instancié à une `liste`, un `int_a` est un terme qui peut être instancié à un `entier`.Dans l'analyseur, ces types peuvent être définis dans le formalisme suivant:

```
anylist(X) ⇔  var(X) @
              X = [] @
              X=[H|T] & anylist(T).

int_a(X) ⇔    var(X) @
              integer(X).
```

Dans le tableau qui suit, il est à remarquer que `[a|al]` est un cas particulier de `anylist` et que `~(a,al)` exprime un *noshar* entre le `anyterm` et la `anylist`.

Les tests sont résumés dans le tableau 1:

N° Test	Pre-post condition in X L LEff	Clause 1	Clause 2	Permutations communes Clause 1
1	g gl al	3	6	-
2	g gl a	8	6	-
3	g gl v	8	6	-
4	g gl [a|al]	0	6	-
5	g gl [a|al], ~(a,al)	0	6	-
6	g gl al a gl al g al gl g gl [a|al]	3 2 3 3	6 6 6 6	0
7	g gl al a gl al g al gl g gl [a|al], ~(a,al)	3 2 3 8	6 6 6 6	0
8	g gl al a gl al g al gl g gl gl a gl al, ~(a,al)	3 2 3 12 2	6 6 6 6 6	0
9	g gl al a gl al g al gl g gl gl a gl a, ~(a,a)	3 2 3 12 4	6 6 6 6 6	0
10	g al gl	3	6	-

Table 1: efface/3

N° Test	Pre-post condition in X L LEff	Clause 1	Clause 2	Permutations communes Clause 1
11	g a gl	8	6	-
12	a gl al	2	6	-
13	a gl a	4	6	-
14	g gl gl	6	6	-
15	g gl al a gl al g al gl g gl gl	3 2 3 12	6 6 6 6	0
16	g gl a a gl a g a gl g gl gl	8 4 8 12	6 6 6 6	2

Table 1: efface/3

5.2.1.3. Commentaires.

Au vu du tableau, la clause 2 nous paraît moins intéressante, puisque pour chacun des tests, les six permutations possibles sont correctes. Ce phénomène est dû au fait que le deuxième ou le troisième paramètre est toujours une `groundlist`. Sans cela, aucune permutation ne serait correcte, à moins d'ajouter un littéral de type checking `list(T)` ou `list(LEff)`. Nous nous attarderons donc plus spécialement à l'étude de la clause 1.
Lors de nos tests, nous avons mis en évidence quatre usages possibles de la procédure `efface/3`.Nous les avons combinés et nous nous avons analysé à quelle(s) condition(s), nous pouvions trouver une procédure `efface/3` multi-directionnelle.
Les quatre usages mis en évidence sont les suivants :

- **Usage 1 (Test 1).**

  in `(groundterm,groundlist,anylist)::`
  out `(groundterm,groundlist,groundlist)`
  \vérification plus calcul d'une liste moins la première occurrence d'un de ses éléments\

- **Usage 2 (Test 10).**

in (groundterm,anylist,groundlist)::
out (groundterm,groundlist,groundlist)
\vérification et calcul de la liste initiale\

- **Usage 3 (Test 12).**

in (anyterm,groundlist,anylist)::
out (groundterm,groundlist,groundlist)
\vérification plus usages variés(usage 1 et 4 notamment)\

- **Usage 4 (Test 14).**

in (groundterm,groundlist,groundlist)::
out (groundterm,groundlist,groundlist)
\vérification\

Dans nos tests réalisés sur `efface/3`, nous avons d'abord calculé le nombre de permutations correctes de chacune des clauses pour chaque usage lorsqu'il était spécifié de façon unique. Nous avons ensuite recherché s'il existait au moins une permutation commune à tous les usages (pour chacune des clauses); la (les) procédure(s) obtenue(s) étant, dans ce cas, multi-directionnelle(s).
Le test 1 nous donne le résultat pour le cas de base (usage 1) tandis que les tests 2 à 5 sont des variantes de ce cas où les contraintes sur les paramètres ont été ou affaiblies (tests 2 et 3) ou renforcées (tests 4 et 5). Lorsque les contraintes sont plus strictes (tests 1, 4 et 5), nous obtenons moins de permutations correctes que lorsque nous permettons que l'argument `LEff` soit `var` ou même `anyterm`. Les contraintes des tests 4 et 5 sont même tellement strictes qu'il n'existe aucune permutation correcte pour la clause 1.En effet, l'appel récursif de `efface/3` demande un usage plus général de la procédure.
Le test 10 (usage 2) permet de trouver trois permutations correctes pour la clause mais aucune n'est commune aux trois trouvées pour le test 1 ni même aux huit trouvées pour les tests 2 et 3.
Le test 12 (usage 3) dérive deux permutations correctes; celles-ci sont incluses dans les trois permutations trouvées pour le test 1 et les huit permutations calculées pour les tests 2 et 3.
Pour le test 14 (usage 4), nous trouvons six permutations correctes mais aucune n'est commune aux permutations des tests 1, 10 et 12. Par contre, cinq sont incluses dans les huit permutations correctes des tests 2 et 3.
Nous venons de tester chaque usage comme étant usage unique de la procédure `efface/3` (uni-directionnalité). Voyons maintenant les résultats obtenus lorsque nous essayons de rendre cette procédure multi-directionnelle. Le test 15 accepte simultanément les quatre usages (tests 1, 10, 12, 14) et, dans ce cas, nous obtenons toujours trois permutations correctes pour l'usage 1 (test 1), trois pour l'usage 2 (test 10), deux pour l'usage 3 (test 12) mais douze (au lieu de six) pour l'usage 4 (test 14). C'est là le signe d'une certaine subtilité de l'analyseur lors de l'analyse d'un appel récursif de `efface/3`. Parmi ces douze permutations, nous retrouvons toutes les permutations correctes pour les autres usages.
Poursuivons notre étude par les tests 6 et 7 et observons plus particulièrement la réaction du quatrième behaviour introduit. Dans le premier cas (test 6), nous n'obtenons que trois permutations correctes, identiques à celles obtenues lors du test 1. Dans le second cas (test 7), huit permutations sont correctes. Cinq

autres permutations sont rendues correctes par l'introduction d'une contrainte de *nosharing*. Nous allons analyser l'une de ces cinq permutations et voir pourquoi cette contrainte supplémentaire permet de la rendre correcte.
Dans le test 6, les trois permutations correctes sont les suivantes :

- **Permutation 1 :**

```
efface (X,L,LEff) :- =(L,[H|T]),not(=(H,X)),efface(X,T,TEff),
                     =(LEff,[H|TEff]).
```

- **Permutation 2 :**

```
efface (X,L,LEff) :- =(L,[H|T]),efface(X,T,TEff),not(=(H,X)),
                     =(LEff,[H|TEff]).
```

- **Permutation 3 :**

```
efface (X,L,LEff) :- =(L,[H|T]),efface(X,T,TEff),=(LEff,[H|TEff]),
                     not(=(H,X)).
```

Dans le test 7, les huit permutations correctes trouvées sont celles-ci :

- **Permutation 1 :**

```
efface (X,L,LEff) :- =(L,[H|T]),not(=(H,X)),efface(X,T,TEff),
                     =(LEff,[H|TEff]).
```

- **Permutation 2 :**

```
efface (X,L,LEff) :- =(L,[H|T]),efface(X,T,TEff),not(=(H,X)),
                     =(LEff,[H|TEff]).
```

- **Permutation 3 :**

```
efface (X,L,LEff) :- =(L,[H|T]),efface(X,T,TEff),=(LEff,[H|TEff]),
                     not(=(H,X)).
```

- **Permutation 4 :**

```
efface (X,L,LEff) :- =(LEff,[H|TEff]),=(L,[H|T]),efface(X,T,TEff),
                     not(=(H,X)).
```

- **Permutation 5 :**

```
efface (X,L,LEff) :- =(LEff,[H|TEff]),=(L,[H|T]),not(=(H,X)),
                     efface(X,T,TEff).
```

- **Permutation 6 :**

```
efface (X,L,LEff) :- =(L,[H|T]),not(=(H,X)),=(LEff,[H|TEff]),
                     efface(X,T,TEff).
```

- **Permutation 7 :**

```
efface (X,L,LEff) :- =(L,[H|T]),=(LEff,[H|TEff]),efface(X,T,TEff),
                     not(=(H,X)).
```

- **Permutation 8 :**

```
efface (X,L,LEff) :- =(L,[H|T]),=(LEff,[H|TEff]),not(=(H,X)),
                     efface(X,T,TEff).
```

Nous allons analyser manuellement - c-à-d sans l'aide de *DETAIL* - la permutation 8 présentée ci-dessus, dans un petit tableau reprenant en ordonnées, les différents paramètres de la clause et en abscisses, les différents points d'exécution de la clause. Les coordonnées du tableau seront le type et le mode calculés de chaque paramètre.Nous avons réalisé ce travail afin de nous rendre compte du temps et de la difficulté demandés par une telle analyse et afin de pouvoir comparer nos résultats à ceux obtenus par l'analyseur. Ce travail est assez fastidieux et pas toujours facile à réaliser manuellement.

```
efface (X,L,LEff) :- 0=(L,[H|T]),
                     1=(LEff,[H|TEff]),
                     2efface(X,T,TEff),
                     3not(=(H,X)).
                     4
```

	0	1	2	3	4
X	g	g	g	g	g

Table 2: Test 6

	0	1	2	3	4
L	gl	[g|gl]	[g|gl]	[g|gl]	[g|gl]
LEff	[a|al]	[a|al]	[g|a]	[g|a]	[g|a]
H	v	g	g	g	g
TEff	v	v	a	a	a
T	v	gl	gl	gl	gl

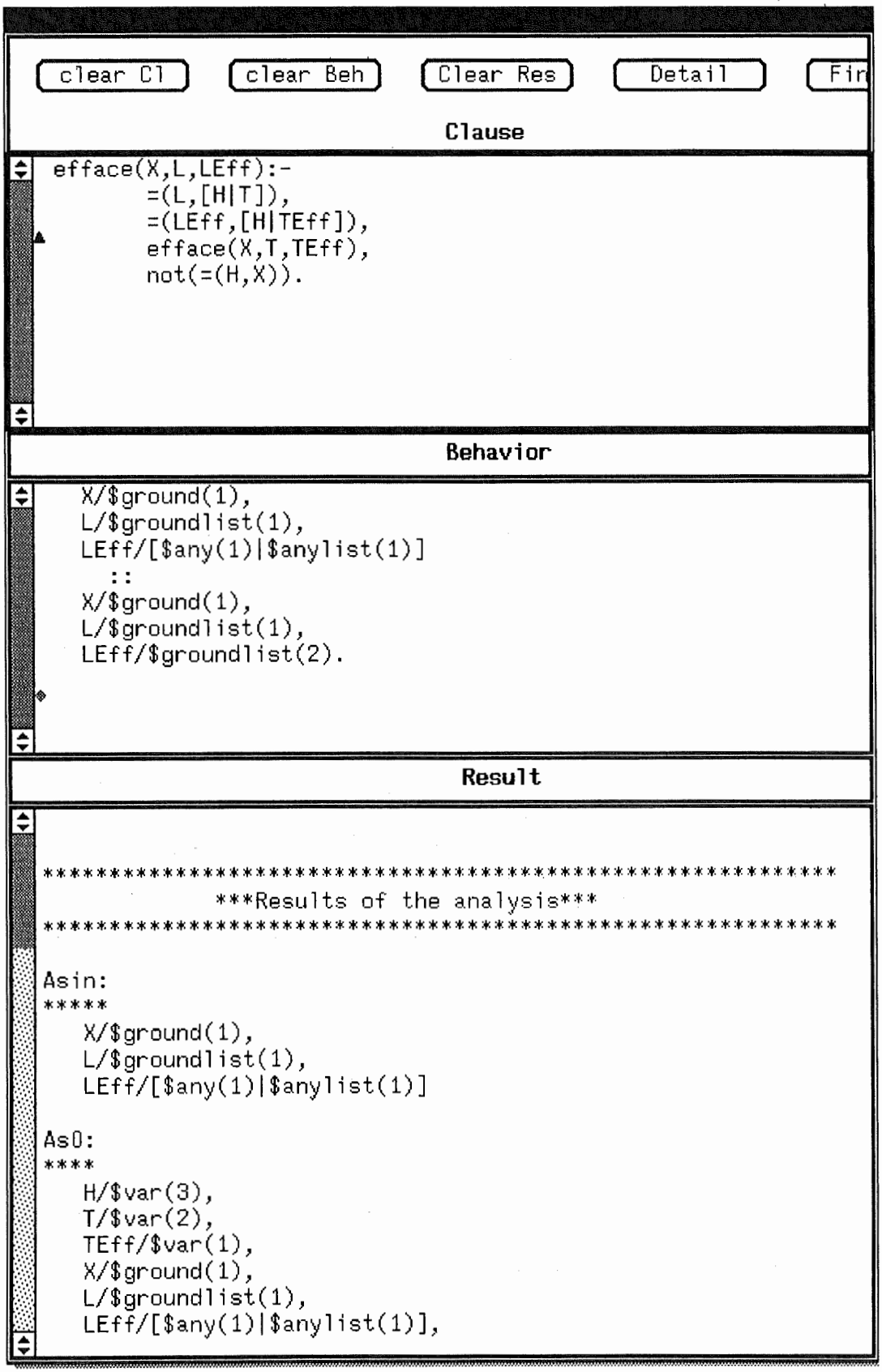
Table 2: Test 6

La partie **out** de la directionnalité de la procédure - **out** (`groundterm,groundlist,groundlist`) - n'est pas respectée et la permutation de la clause est donc incorrecte.

	0	1	2	3	4
X	g	g	g	g	g
L	gl	[g|gl]	[g|gl]	[g|gl]	[g|gl]
LEff	[a|al]	[a|al]	[g|al]	[g|gl]	[g|gl]
H	v	g	g	g	g
TEff	v	v	al	al	al
T	v	gl	gl	gl	gl

Table 3: Test 7

Dans ce cas, la partie **out** de la directionnalité est bien respectée et la permutation de la clause est correcte.
Le point névralgique de l'analyse se situe au point de substitution 2. Dans le premier cas, `TEff` devient `any` tandis que dans le deuxième, `TEff` devient `anylist`; ce qui permet d'obtenir un appel correct de `efface(X,T,TEff)`.
En effet, lors de l'appel de =`(LEff,[H|TEff])` avec `LEff` étant `[any|anylist]` et `[H|TEff]` étant `[ground|var]`, nous ne pouvons conclure que `TEFF` est `anyterm` (test 6) car il n'existe aucune contrainte de *nosharing* dans `LEff`. Nous pourrions obtenir `LEff=[X|X]` comme substitution, `H` étant `ground` et supposé instancié au terme ground `a`, nous pourrions obtenir `LEff=[a|a]` et l'appel de la clause `efface(X,T,TEff)`, avec `T` étant une `groundlist` (par exemple `[]`), deviendrait `efface(a,[],a)` et serait incorrect. Le *noshar* introduit (test 7) exclut cette situation.
Ces calculs manuels, qui peuvent devenir fastidieux pour des procédures moins classiques sont confirmés par l'utilisation de l'outil *DETAIL* , comme le montrent les écrans (test 6) ci-après.


clear Cl
clear Beh
Clear Res
Detail
Fir
Clause
efface(X,L,LEff):-
=(L,[H|T]),
=(LEff,[H|TEff]),
efface(X,T,TEff),
not(=(H,X)).
Behavior
X/$ground(1),
L/$groundlist(1),
LEff/[$any(1)|$anylist(1)]
::
X/$ground(1),
L/$groundlist(1),
LEff/$groundlist(2).
Result
Results of the analysis
Asin:
*****
X/$ground(1),
L/$groundlist(1),
LEff/[$any(1)|$anylist(1)]
As0:
****
H/$var(3),
T/$var(2),
TEff/$var(1),
X/$ground(1),
L/$groundlist(1),
LEff/[$any(1)|$anylist(1)],

Result

```
   LEff/[$any(1)|$anylist(1)],
   ~($var(3),$any(1)),
   ~($var(3),$anylist(1)),
   ~($var(3),$var(1)),
   ~($var(3),$var(2)),
   ~($var(2),$any(1)),
   ~($var(2),$anylist(1)),
   ~($var(2),$var(1)),
   ~($var(1),$any(1)),
   ~($var(1),$anylist(1))

      **call: =(L,[H|T])**   (consistency ensured)

As1:
****
   H/$ground(5),
   T/$groundlist(5),
   TEff/$var(2),
   X/$ground(2),
   L/[$ground(5)|$groundlist(5)],
```

Result

```
   L/[$ground(5)|$groundlist(5)],
   LEff/[$any(7)|$anylist(2)],
   ~($var(2),$any(7)),
   ~($var(2),$anylist(2))

      **call: =(LEff,[H|TEff])**   (consistency ensured)

As2:
****
   H/$ground(14),
   T/$groundlist(9),
   TEff/$any(22),
   X/$ground(4),
   L/[$ground(14)|$groundlist(9)],
   LEff/[$ground(14)|$any(22)]

      **call: efface(X,T,TEff)**   (consistency is not ensur
)

As3:
```

Result

```
As3:
****
   H/$ground(14),
   T/$groundlist(9),
   TEff/$any(23),
   X/$ground(4),
   L/[$ground(14)|$groundlist(9)],
   LEff/[$ground(14)|$any(23)]

      **call: not(=(H,X))**   (consistency ensured)

As4:
****
   H/$ground(18),
   T/$groundlist(2),
   TEff/$any(25),
   X/$ground(5),
   L/[$ground(18)|$groundlist(2)],
   LEff/[$ground(18)|$any(25)]
```

Result

```
   LEff/[$ground(18)|$any(25)]

Asout_final:
************
   X/$ground(5),
   L/[$ground(18)|$groundlist(2)],
   LEff/[$ground(18)|$any(25)]

(Consistency is not ensured for the Asout_final)
```

Poursuivons notre analyse et élargissons le domaine d'application de la procédure en supprimant les contraintes sur les paramètres présentes dans les tests 1, 10 et 12. Nous obtenons alors respectivement les `prepost` 2, 11 et 13. De trois permutations correctes pour le test 1, nous passons à huit; de même pour le test 10 tandis que pour le test 12, nous passons à quatre permutations correctes.
Le test 16 permet d'obtenir 2 permutations correctes communes. Cette libéralisation des contraintes sur les paramètres permet donc d'obtenir une procédure véritablement multi-directionnelle.

Les deux permutations correctes obtenues sont les suivantes:

- **Permutation commune 1 :**

```
efface (X,L,LEff) :- =(LEff,[H|TEff]),=(L,[H|T]),efface(X,T,TEff),
                     not(=(H,X)).
```

- **Permutation commune 2 :**

```
efface (X,L,LEff) :- =(L,[H|T]),=(LEff,[H|TEff]),efface(X,T,TEff),
                     not(=(H,X)).
```

Terminons cet exemple avec deux conclusions plus générales.
Premièrement, nous pouvons relever toute la subtilité de l'analyseur lorsque le nombre d'usages augmente. En effet, lors du test 14, nous avons obtenu six permutations correctes pour ce behaviour tandis que nous en avons obtenu douze lors du test 15 (Il n'était plus usage unique).Ceci s'explique par le fait que, lors de l'appel récursif de `efface/3`, plus de possibilités sont offertes sur les types et les modes des paramètres en entrée.
Deuxièmement, nous avons pu constater qu'une libéralisation des contraintes sur les paramètres nous avait permis d'obtenir une procédure véritablement multi-directionnelle. Nous pouvons alors douter de la nécessité du type `anytype`; celui-ci pouvant être remplacé, de meilleure façon, par `anyterm`. En effet, pour une procédure `efface/3` inchangée, nous obtenons plus de permutations correctes et des permutations communes aux quatre usages définis. Le type checking implicite exprimé par `anytype` ne nous apparaît donc pas comme strictement nécessaire.

Nous allons tâcher maintenant de vérifier ces constatations sur d'autres procédures: `member/2` (section 5.2.2.), `length/2` (section 5.2.3.), `reverse/2` (section 5.2.4), `append/3` (section 5.2.5.).

5.2.2. CAS 2 : *member/2.*

5.2.2.1. Spécification et description logique.

La procédure `member/2` est construite pour vérifier si un élément appartient à une liste ou à obtenir un élément de la liste.

Sa spécification est la suivante :

procédure `member (X,L)`

Type : `X: term`
`L: list`

Relation: `X` est un élément de `L`.

Conditions d'application:

in `(any,ground)::` **out** `(ground,ground)<0-n>`
\obtenir un élément de la liste\

in `(ground,ground)::` **out** `(ground,ground)<0-1>`
\vérifier si un élément appartient à la liste\

in `(ground,any)::` **out** `(ground,any)<0-n>`
\vérification et génération de liste\

La description logique prend la forme suivante :

```
member (X,L) ⇔        L=[] ∧ false ∨
                      L=[H|L1] ∧ (H=X ∨ ~(H=X) ∧ member(X,L1)).
```

5.2.2.2. Tests et résultats.

Lors de la dérivation syntaxique, nous obtenons les clauses suivantes:

- **Clause 1 :**

```
member (X,L) ←        not(=(H,X)),member(X,L1),=(L,[H|L1]).
```

- **Clause 2 :**

```
member (X,L) ←        =(H,X),=(L,[H|L1]).
```

Le tableau 4 résume les tests de ces deux clauses:

N° Test	Pre-post condition in X	Pre-post condition in L	Clause 1	Clause 2
1	in: a out: g	gl gl	1	2
2	in: g out: g	gl gl	2	2
3	in: g out: g	al al	0	0
4	in: g out: g	al gl	0	0
5	in: g out: g	a al	0	0

Table 4: member/2

Pour les deux premiers usages, il existe une permutation correcte commune:

```
member (X,L) :-        =(L,[H|L1]),member(X,L1),not(=(H,X)).
```

Pour le deuxième usage, la permutation suivante est également correcte:

```
member (X,L) :-        =(L,[H|L1]),not(=(H,X)),member(X,L1).
```

5.2.2.3. Commentaires.

Pour le premier usage, il est nécessaire que l'unification soit exécutée d'abord et ensuite l'appel récursif de `member/2` pour qu'enfin, le dernier littéral (`not/1`) soit appelé correctement avec des arguments `ground`. Pour le troisième usage, il n'existe aucune permutation correcte pour la clause 1. Cela est dû au littéral `not(H=X)` qui demande un ground goal. De même, la clause 2 n'a aucune permutation correcte car un *sharing* interne dans `L` peut l'instancier à un mauvais type. Une solution pour la clause 2 est d'interdire ce *sharing* (par exemple en demandant que `L` soit `var`). Quant à la clause 1, une solution est de supprimer le `not/1` qui n'a aucune utilité ici. En effet, les descriptions logiques de `member/2` avec ou sans `not/1` sont équivalentes.
Dans les tests 4 et 5, nous demandons que `L` soit une `anylist` en sortie. Le but de la procédure, avec cet usage est de générer une liste `L` d'une certaine longueur dont les éléments sont des variables. Le test 4 et le test 5 sont différents car dans le premier, nous demandons que `L` soit une `anylist` et dans le second, nous

demandons qu'elle soit un `anyterm` en entrée. Dans chacun des cas et pour chacune des clauses, nous ne trouvons de permutation correcte.Dans la clause 1, la suppression du `not/1` et l'interdiction d'un *sharing* interne à `L` permettrait de trouver une solution. Une solution pour la clause 2 est également d'interdire ce *sharing* interne.

5.2.3. CAS 3 : *length/2.*

5.2.3.1. Spécification et description logique.

La procédure `length/2` est conçue pour calculer la longueur `N` d'une liste d'éléments `L`.

Nous pouvons la spécifier de la façon suivante :

procédure `length (L,N)`

Type : `N: positive integer`
`L: list`

Relation: `N` est le nombre d'éléments de `L`.

Conditions d'application:

in `(ground,any)::`**out** `(ground,ground)<0-1>`
\calculer la longueur d'une liste\
\vérifier la longueur d'une liste\

in `(any,ground)::`**out** `(any,ground)<0-n>`
\générer une liste d'une certaine longueur (avec des variables pour éléments)\

Sa description logique est la suivante :

```
length (L,N) ⇔      L=[] ∧ N=0 ∨
                    L=[H|T] ∧ length(T,N_T)) ∧ N is N_T + 1.
```

5.2.3.2. Tests et résultats.

Avant de présenter le tableau des tests, précisons les `prepost` que nous avons spécifiées pour `is/2`:

in `(var,athexpr)::`**out** `(integer,athexpr)<1-1>`
in `(integer,athexpr)::`**out** `(integer,athexpr)<0-1>`

`is (N,E)` consiste à évaluer l'expression arithmétique `E` dans `N`.

Lors de la dérivation syntaxique de `length/2`, nous obtenons les clauses suivantes:

- **Clause 1 :**

```
length (L,N) ←        length(T,N_T),is(N,+(N_T,1)),=(L,[H|T]).
```

- **Clause 2 :**

```
length (L,N) ←        =(L,[]),=(N,0).
```

Le tableau 5 reprend les tests effectués pour `length/2`:

N° Test	Pre-post condition in L	N	Clause 1	Clause 2
1	gl	int_a	1	2
2	gl	a	0	2
3	al	g	0	2
4	a	g	0	2
5	gl al	int_a g	1 0	2 2

Table 5: length/2

5.2.3.3. Commentaires.

Dans le test 2, aucune permutation correcte n'existe pour la clause 1 car aucune permutation ne rend possible un appel correct de `is/2`, `n` étant `any`.
Lors des tests 3 et 4, nous ne trouvons pas de permutations correctes pour la clause 1. En effet, il n'est pas possible que l'appel récursif de `length/2` se fasse correctement du fait des conditions d'application de la procédure `is/2`. Pour que `N_T` soit `ground` au moment de l'appel de `length/2`, il faudrait que `is/2` soit capable, à partir d'un entier `N`, de calculer une expression arithmétique dont il résulte.

5.2.4. CAS 4 : *reverse/2.*

5.2.4.1. Spécification et description logique.

La procédure `reverse/2` est conçue pour calculer la liste inversée `LRev` d'une liste `L`.

Sa spécification est la suivante :

procédure `reverse (L,LRev)`

Type : `L,LRev: list`

Relation: `LRev` est la liste `L` inversée.

Conditions d'application:

in `(ground,any)::`**out** `(ground,ground)<0-1>`
\trouver la liste `L` inversée\

in `(any,ground)::`**out** `(ground,ground)<0-1>`
\trouver la liste originale `L` à partir de `LRev`\

Nous obtenons alors la description logique suivante :

```
reverse (L,LRev) ⇔   L=[] ∧ LRev=[] ∨
                     L=[H|T] ∧ reverse(T,Rem_LRev) ∧
                     append(Rem_LRev,[H],LRev).
```

5.2.4.2. Tests et résultats.

Avant de présenter le tableau des tests pour `reverse/2`, précisons les `prepost` spécifiées pour `append/3`:

in `(ground,ground,any)::`**out** `(ground,ground,ground)<0-1>`
in `(any,any,ground)::`**out** `(ground,ground,ground)<0-n>`

`append(L1,L2,L3)` consiste à concaténer les listes `L1` et `L2` en la liste `L3`.

La procédure dérivée pour `reverse/2` est la suivante:

- **Clause 1 :**

```
reverse (L,LRev) ←   reverse(T,Rem_LRev),
                     append(Rem_LRev,[H],LRev),
                     =(L,[H|T]).
```

- **Clause 2 :**

```
reverse (L,LRev) ←   =(L,[]),
```

```
=(LRev,[]).
```

Le tableau 6 des tests de `reverse/2` est le suivant:

N° Test	Pre-post condition in L	LRev	Clause 1	Clause 2	P.C. Clause 1	P.C.[1] Clause 2
1	gl	al	1	2	-	-
2	al	gl	1	2	-	-
3	gl al	al gl	1 1	2 2	0	2
4	(1) gl a	a gl	1 3	2 2	0	2
5	(2) gl a	a gl	0 3	2 2	0	2

Table 6: reverse/2

1. P.C. signifie Permutation Commune

5.2.4.3. Commentaires.

(1) et (2) : Les tests 4 et 5 différent en ce sens que la directionnalité de la procédure `append/3` est moins contraignante pour le test 4 (`anyterm` au lieu de `anylist`).

Pour la clause 1 et la directionnalité 1, dans les tests 1, 3 et 4, seule la permutation suivante est correcte:

```
reverse (L,LRev) :-   =(L,[H|T]),
                      reverse(T,Rem_LRev),
                      append(Rem_LRev,[H],LRev).
```

Pour la clause 1 et la directionnalité 2, les tests 2 et 3 permettent de trouver une seule permutation correcte:

```
reverse (L,LRev) :-   append(Rem_LRev,[H],LRev),
                      reverse(T,Rem_LRev),
                      =(L,[H|T]).
```

La libéralisation des contraintes sur les paramètres, dans le test 4, permet de trouver trois permutations correctes pour ce behaviour:

```
reverse (L,LRev) :-   append(Rem_LRev,[H],LRev),
                      =(L,[H|T],
                      reverse(T,Rem_LRev).

reverse (L,LRev) :-   append(Rem_LRev,[H],LRev),
                      reverse(T,Rem_LRev),
                      =(L,[H|T].

reverse (L,LRev) :-   =(L,[H|T],
                      append(Rem_LRev,[H],LRev),
                      reverse(T,Rem_LRev).
```

Ces trois permutations sont également celles trouvées lors du test 5.
Seule une libéralisation des contraintes de type a permis d'obtenir plus de permutations correctes pour la clause 1. Cependant, cela n'est pas suffisant pour obtenir une permutation correcte commune aux deux usages pour cette clause car l'usage 1 demande que l'appel de `reverse/2` soit effectué avant celui de `append/3` et l'usage 2 exige l'inverse.

5.2.5. CAS 5 : *append/3*.

5.2.5.1. Spécification et description logique.

La procédure `append/3` est construite pour concaténer deux listes `L1` et `L2` en une troisième liste `L3`.

Sa spécification est la suivante :

procédure `append (L1, L2, L3)`

Type : `L1,L2,L3 : list`

Relation: { `(L1,L2,L3)` : `L3` est la liste résultant de la concaténation de `L1` et `L2`.}

Conditions d'application:

in `(ground,ground,any)` : : **out** `(ground,ground,ground)` `<0-1>`
\vérification et concaténation\

in `(any,any,ground)` : : **out** `(ground,ground,ground)` `<0-n>`
\vérification, calcul de début (fin) d'une liste étant donné son suffixe (préfixe), éclater une liste en deux parties\

La description logique s'écrit comme suit :

```
append (L1,L2,L3) ⇔   L1=[] ∧ L2=L3 ∨
                      L1=[H|T] ∧ append(T,L2,Rem_L3) ∧
```

```
L3=[H|Rem_L3].
```

5.2.5.2. Tests et résultats.

Cette description logique est traduite syntaxiquement en les deux clauses suivantes:

- **Clause 1 :**

```
append (L1,L2,L3) ←  append(T,L2,Rem_L3),
                     =(L3,[H|Rem_L3]),
                     =(L1,[H|T]).
```

- **Clause 2 :**

```
append (L1,L2,L3) ←  =(L1,[]),
                     =(L2,L3).
```

Le tableau 7 des tests effectués se présente comme suit:

N° Test	Pre-post condition in L1	L2	L3	Clause 1	Clause 2	P.C. Clause 1	P.C. Clause 2
1	gl	gl	al	1	2	0	2
	al	al	gl	1	2		
2	gl	gl	a	3	2	2	2
	a	a	gl	3	2		

Table 7: append/3

5.2.5.3. Commentaires.

Pour la première directionnalité du test 1, seule la permutation suivante est correcte:

```
append (L1,L2,L3) :-  =(L1,[H|T]),
                      append(T,L2,Rem_L3),
                      =(L3,[H|Rem_L3]).
```

La deuxième directionnalité de ce même test accepte la permutation suivante:

```
append (L1,L2,L3) :-  =(L3,[H|Rem_L3]),
```

```
                    append(T,L2,Rem_L3),
                    =(L1,[H|T]).
```

Si nous élargissons l'ensemble des appels permis de la procédure, en diminuant les contraintes sur les paramètres en entrée, nous obtenons deux permutations supplémentaires (communes) pour les deux behaviours:

```
append (L1,L2,L3) :-  =(L3,[H|Rem_L3]),
                      =(L1,[H|T]),
                      append(T,L2,Rem_L3).

append (L1,L2,L3) :-  =(L1,[H|T]),
                      =(L3,[H|Rem_L3]),
                      append(T,L2,Rem_L3).
```

5.3. UTILISATION LORS DE LA REALISATION D'UNE APPLICATION SIMPLE DE BASE DE DONNEES.

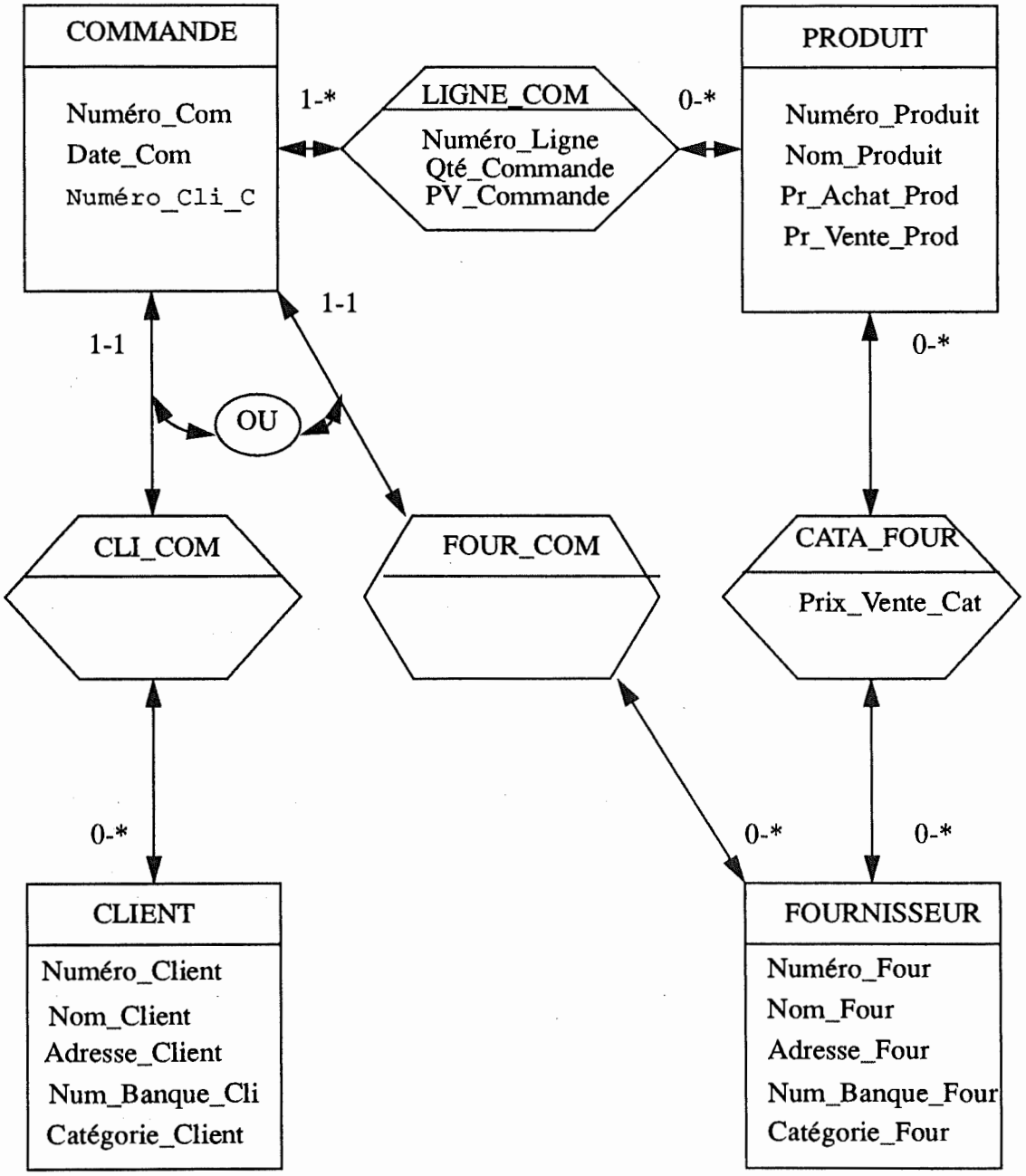
Dans cette troisième partie du chapitre, nous allons appliquer la méthodologie à un cas sortant des procédures PROLOG traditionnelles. Nous lui appliquerons également le schéma d'analyse soumis aux procédures de la section précédente et nous en tirerons des enseignements que nous comparerons aux conclusions de la section 5.2.
Cette section 5.3. va être divisée en deux grandes sous-sections; la première présentant le cas à étudier -une gestion de commandes clients et fournisseurs dans une entreprise dont l'activité consiste en l'achat et la vente de produits-; la deuxième partie consistant en l'étude d'une query particulière, relative à ce cas, selon le même principe que les procédures de la section 5.2.

5.3.1. PRESENTATION DU CAS.

Ce cas est inspiré des exemples présentés dans le cours "Conception de Bases de Données" de Mr J.-L. Hainaut. Dans un premier temps, nous allons le présenter de façon informelle. Nous le spécifierons ensuite de manière relationnelle puis nous définirons les types dont nous avons besoin et enfin, nous présenterons quelques questions possibles parmi lesquelles, nous en choisirons une à analyser.

5.3.1.1. Gestion des commandes clients et fournisseurs.

L'activité de l'entreprise (expl:commerce de vêtements) se limite à acheter des produits aux fournisseurs et à les revendre aux clients. Nous désirons gérer automatiquement nos commandes fournisseurs et clients ainsi qu'obtenir des informations sur nos fournisseurs et sur nos clients. Les commandes fournisseurs se font sur base d'un catalogue.
Le schéma Entité-Association de l'application est présenté ci-après:



Dans ce schéma, nous distinguons quatre entités: **CLIENT**, **FOURNISSEUR**, **PRODUIT**, **COMMANDE** et quatre associations: **LIGNE_COM**, **CLI_COM**, **FOUR_COM** et **CATA_FOUR**.

L'entité **CLIENT** possède les attributs suivants:

`Numéro_Client` : son numéro d'identification dans nos fichiers

`Nom_Client` : le nom du client

Adresse_Client : son adresse (Rue,Code Postal, Localité)
Num_Banque_Cli : son muméro de compte bancaire
Catégorie_Client : la catégorie dans laquelle nous l'avons classé

L'entité **FOURNISSEUR** possède les attributs suivants:

Numéro_Four : son numéro d'identification dans nos fichiers
Nom_Four : le nom du fournisseur
Adresse_Four : son adresse (Rue,Code Postal, Localité)
Num_Banque_Four : son muméro de compte bancaire
Catégorie_Four : la catégorie dans laquelle nous l'avons classé

L'entité **PRODUIT** possède les attributs suivants:

Numéro_Produit : son numéro d'identification dans nos fichiers
Nom_Produit : le nom du produit
Pr_Achat_Prod : prix d'achat normal du marché
Pr_Vente_Prod : prix de vente normal du marché

L'entité **COMMANDE** possède les attributs suivants:

Numéro_Com : son numéro d'identification dans nos fichiers
Numéro_Cli_C : le numéro du client qui a passé commande
Date_Com : la date à lquelle la commande a été passée

L'association **LIGNE_COM** possède les attributs suivants:

Numéro_Ligne : le numéro d'identification de la ligne dans la commande
Qté_Commande : la quantité du produit commandée
PV_Commande : le prix auquel le produit a été commandé

Une commande contient une ou plusieurs lignes de commande. Une ligne de commande appartient à une seule commande.

L'association **CATA_FOUR** possède les attributs suivants:

PV_Produit_Fourn : le prix de vente renseigné dans le catalogue par le fournisseur

Un fournisseur possède 0, une ou plusieurs entrées au catalogue. Un produit peut être répertorié 0, une ou plusieurs fois au catalogue.

Les associations **CLIENT_COMMANDE** et **FOURNISSEUR_COMMANDE** ne possèdent pas d'attributs.
Un client (fournisseur) possède 0, une ou plusieurs commandes. Une commande appartient à un et un seul client (fournisseur).

5.3.1.2. Schéma relationnel de l'application.

Dans cette application de gestion des commandes, nous trouvons la nécessité de présenter 6 relations : client, fournisseur, produit, catalogue, commande, ligne_commande.

Le domaine de chaque attribut sera défini dans la sous-sous-section suivante.
Dans chaque relation, l'item ou le groupe d'items souligné est l'identifiant de la relation.

client(Numéro, Nom, Adresse, Numéro-Banque, Catégorie).

fournisseur(Numéro, Nom, Adresse, Numéro-Banque, Catégorie).

produit(Numéro, Nom, Prix-Achat, Prix-Vente).

catalogue(Numéro-Produit, Numéro-Fournisseur, Prix-Vente).

commande(Numéro-Commande-C/F, Numéro-C/F, Date-Commande).

ligne_com(Numéro-Com-C/F,Numéro-Ligne-Com-C/F,Numéro-Prod,Qté-Com,Prix-Vente-Com).

5.3.1.3. Définition des types.

Les types dont nous avons besoin sont définis ci-dessous. Avant leur lecture précisons ce qui suit:
les types él_* représentent les éléments d'une table *_T. (Par exemple,él_client représente un client de la table client_T.)

```
él_client(X) ⇔      X = él_client(Numéro,Nom,Adresse,Numéro-Banque,Catégorie) &
                    integer(Numéro) &
                    atom(Nom) &
                    adresse(Adresse) &
                    atom(Numéro-Banque) &
                    atom(Catégorie).

client_T(X) ⇔       X = [] @
                    X=[H|T] & él_client(H) & client_T(T).

adresse(X) ⇔        X = adresse(Rue,Code-Postal,Localité) &
                    atom(Rue) &
                    integer(Code-Postal) &
                    atom(Localité).

él_fournisseur(X) ⇔ X = él_fournisseur(Numéro,Nom,Adresse,Numéro-Banque,Catégorie) &
                    integer(Numéro) &
                    atom(Nom) &
                    adresse(Adresse) &
                    atom(Numéro-Banque) &
                    atom(Catégorie).

fournisseur_T(X) ⇔ X = [] @
                    X=[H|T] & él_fournisseur(H) & fournisseur_T(T).
```

```
él_produit(X) ⇔     X = él_produit(Numéro,Nom,Prix-Achat,Prix-Vente) &
                    integer(Numéro) &
                    atom(Nom) &
                    integer(Prix-Achat) &
                    integer(Prix-Vente).

produit_T(X) ⇔      X = [] @
                    X=[H|T] & él_produit(H) & produit_T(T).

él_cata(X) ⇔        X = él_cata(Numéro-Produit,Numéro-Fournisseurt,Prix-Vente) &
                    integer(Numéro-Produit) &
                    integer(Numéro-Fournisseur) &
                    integer(Prix-Vente).

catalogue_T(X) ⇔    X = [] @
                    X=[H|T] & él_cata(H) & catalogue_T(T).

él_commande(X) ⇔ X = él_commande(Numéro-Commande-C/F,Numéro-C/F,Date-Commande) &
                    integer(Numéro-Commande-C/F) &
                    integer(Numéro-C/F) &
                    atom(Date-Commande).

commande_T(X) ⇔ X = [] @
                    X=[H|T] & él_commande(H) & commande_T(T).

él_ligne_com(X) ⇔ X=él_ligne_com(Numéro-Com-C/F,Numéro-Ligne-Com-C/F,Numéro-Prod,Qté-
Com,Prix-Vente-Com) &
                    integer(Numéro-Com-C/F) &
                    integer(Numéro-Ligne-Com-C/F) &
                    integer(Numéro-Prod)&
                    integer(Qté-Com)&
                    integer(Prix-Vente-Com).

ligne_com_T(X) ⇔ X = [] @
                    X=[H|T] & él_ligne_com(H) & ligne_com__T(T).
```

5.3.1.4. Quelques questions possibles.

Dans une entreprise de ce type, quelques statistiques sont nécessaires et les questions sélectionnées ci-après peuvent être utiles:

1) Dans le catalogue des produits, il est utile de connaître, pour un certain produit, le ou les fournisseurs les plus avantageux.

2) Il est également utile de connaître tous les fournisseurs d'un produit donné, ceci afin d'éviter des ruptures de stock.

3) Il est parfois nécessaire, afin d'organiser des mailings, de connaître les clients, les fournisseurs d'une ville, d'une catégorie.

4) Afin de gérer notre stock, nous serons amenés à nous demander quels sont les produits commandés (clients et fournisseurs) et en quelles quantités.

Bien d'autres questions pourraient encore être posées. C'est la première de cette série que nous avons sélectionnée afin de l'étudier. Nous allons implémenter la procédure de cette query grâce à la méthodologie de Y. Deville et nous allons l'analyser de façon analogue aux procédures de la section 5.2. grâce à l'analyseur.

5.3.2. LA PROCEDURE MEILLEUR_PRIX/4.

5.3.2.1. Spécification et description logique.

A la base, cette procédure est conçue pour rechercher dans un catalogue, le ou les fournisseurs qui appliquent le meilleur prix de vente pour un produit donné.

Sa spécification est la suivante :

procédure `meilleur_prix (P,PV,F,CAT)`

Type : `P`[1]`,PV,F`[2] `: integer`
`CAT : catalogue_T`

Relation: L'ensemble des triplets qui appartiennent au catalogue et tels que, pour un même produit, il n'existe pas de fournisseur ayant un meilleur prix.

Conditions d'application:

in `(ground,any,any,ground) : :` **out** `(ground,ground,ground,ground) <0-n>`
\rechercher le ou les meilleurs fournisseurs d'un produit\

in `(any,any,ground,ground) : :` **out** `(ground,ground,ground,ground) <0-n>`
\rechercher le ou les produits vendus à meilleur prix par un fournisseur\

in `(ground,ground,ground,ground) : :`
out `(ground,ground,ground,ground) <0-1>`
\vérification\

---

1. P est entier et identifiant de él_produit.
2. F est entier et identifiant de él_pournisseur.

in (ground, any, ground, ground) : :
out (ground, ground, ground, ground) <0-n>
\vérifier si un certain fournisseur vend un certain produit à meilleur prix et donner celui-ci\

in (any, any, any, ground) : :**out** (ground, ground, ground, ground) <0-n>
\donner le catalogue complet des meilleures offres\

La description logique de `meilleur_prix/4` est la suivante :

```
meilleur_prix (P,PV,F,CAT) ⇔ CAT=[él_cata(P,PV,F)|CAT_FT] ∧
                             test_mp(P,PV,CAT_FT) ∨
                             CAT=[él_cata(P,PV1,F1)|CAT_FT] ∧
                             PV1>=PV ∧
                             meilleur_prix(P,PV,F,CAT_FT)∨
                             CAT=[él_cata(P1,PV1,F1)|CAT_FT] ∧
                             P1<>P ∧
                             meilleur_prix(P,PV,F,CAT_FT).
```

Nous allons commenter brièvement la réflexion qui a produit cette description logique en expliquant chacune des conjonctions.
Ou bien le triplet (P,PV,F) s'unifie avec le premier élément du catalogue et l'on vérifie s'il n'en existe pas de meilleur.
Ou bien P s'unifie avec le produit du premier élément du catalogue et si son prix est inférieur à celui-ci, on vérifie dans le reste du catalogue s'il n'en existe pas de meilleur.
Ou bien P ne s'unifie pas avec le produit du premier élément du catalogue et l'on vérifie dans le reste du catalogue s'il n'en existe pas de meilleur.

Pour des besoins futurs, nous donnerons également la spécification et la description logique de la procédure `test_mp/3`:

procédure `test_mp (P,PV,CAT)`

Type : `P`[1]`,PV : integer`
`CAT : catalogue_T`

Relation: La procédure vérifie si le couple (P,PV) est le meilleur du catalogue.

Conditions d'application:

in (ground, ground, ground) : :**out** (ground, ground, ground) <0-1>
\vérification\

```
test_mp (P,PV,CAT) ⇔          CAT=[] ∨
```

1. P est entier et identifiant de él_produit.

```
CAT=[él_cata(P,PV1,F1)|CAT_FT] ∧
PV1>=PV ∧
test_mp(P,PV,CAT_FT)∨
CAT=[él_cata(P1,PV1,F1)|CAT_FT] ∧
P1<>P ∧
test_mp(P,PV,CAT_FT).
```

5.3.2.2. Tests et résultats.

Les trois clauses dérivées de la description logique sont les suivantes:

- **Clause 1 :**

```
meilleur_prix(_P,_PV,_F,_CAT):-_CAT=[él_(_P1,_PV1,_F1)|_CAT_FT],
                               meilleur_prix(_P,_PV,_F,_CAT_FT),
                               not(_P1=_P).
```

- **Clause 2 :**

```
meilleur_prix(_P,_PV,_F,_CAT):-_CAT=[él_cata(_P,_PV1,_F1)|_CAT_FT],
                               meilleur_prix(_P,_PV,_F,_CAT_FT),
                               geq(_PV1,_PV).
```

- **Clause 3 :**

```
meilleur_prix(_P,_PV,_F,_CAT):-_CAT=[él_cata(_P,_PV,_F)|_CAT_FT],
                               test_mp(_P,_PV,_CAT_FT).
```

Le tableau 6 présente les tests effectués avec contraintes de types:

Pre-post condition in P	PV	F	CAT	Clause 1	Clause 2	Clause 3	P.C. Clause 1	P.C. Clause 2	P.C. Clause 3
g	i_a[1]	i_a	g	2	1	1			
i_a	i_a	g	g	1	0	1	1	0	1
g	i_a	g	g	2	1	1			
g	g	g	g	2	2	1			

Table 8: meilleur_prix/4 (avec c.t.)

1. i_a signifie int_any

Avant de commenter briévement ce tableau, faisons deux remarques :

1) CAT est toujours ground et de type catalogue_T.

2) la procédure test_mp/3 a comme directionnalité :
in (ground,ground,ground) : :**out** (ground,ground,ground) <0-1>

Nous constatons dans le tableau 6 ci-dessus qu'il n'existe aucune permutation correcte commune pour la clause 2. Seule l'introduction d'un *noshar* entre P et PV permettrait de trouver une permutation pour cette directionnalité. Cette permutation serait commune à celle des autres directionnalités.En effet, pour le deuxième usage, _PV deviet any et l'appel de meilleur_prix est incorrect. Ici, l'analyseur pourrait, parce que _P et _PV sont même type, déduire que _PV est any_int et rendre ainsi l'appel correct. Cette légère imprécision ne se produit que dans ce cas et parce qu'il n'existe pas de *noshar* entre _P et _PV.

Le tableau 8 présente les tests effectués sans contraintes de types:

Pre-post condition in P	PV	F	CAT	Clause 1	Clause 2	Clause 3	P.C. Clause 1	P.C. Clause 2	P.C. Clause 3
g	a	a	g	2	1	1			
a	a	g	g	1	1	1	1	1	1
g	a	g	g	2	1	1			
g	g	g	g	2	2	1			

Table 9: meilleur_prix/4 (sans c.t.)

Dans cette optique, nous trouvons une permutation correcte pour chaque clause.
A partir de ces constatations, nous avons écrit une procédure PROLOG et nous l'avons appliquée à un petit catalogue.
La procédure écrite est la suivante:

```
meilleur_prix(_P,_PV,_F,_CAT):-_CAT=[él_(_P,_PV,_F)|_CAT_FT],
                               test_mp(_P,_PV,_CAT_FT).

meilleur_prix(_P,_PV,_F,_CAT):-_CAT=[él_cata(_P,_PV1,_F1)|_CAT_FT],
                               meilleur_prix(_P,_PV,_F,_CAT_FT),
                               geq(_PV1,_PV).

meilleur_prix(_P,_PV,_F,_CAT):-_CAT=[él_cata(_P1,_PV1,_F1)|_CAT_FT],
                               meilleur_prix(_P,_PV,_F,_CAT_FT),
                               not(_P1=_P).

test_mp(_P,_PV,_CAT):-         _CAT=[].
```

```
test_mp(_P,_PV,,_CAT):-        _CAT=[él_cata(_P,_PV1,_F1)|_CAT_FT],
                               test_mp(_P,_PV,_CAT_FT),
                               geq(_PV1,_PV).

test_mp(_P,_PV,_CAT):-         _CAT=[él_cata(_P1,_PV1,_F1)|_CAT_FT],
                               test_mp(_P,_PV,_CAT_FT),
                               not(_P1=_P).

geq(_PV1,_PV):-                _PV1>=_PV.
```

Le catalogue suivant a été choisi:

{(10,1500,1),(20,2000,2),(30,1800,3),(10,1000,3),(40,850,2),(50,1750,1),(40,700,1),(20,1800,1),(50,1800,3),(20,1950,3)(10,1000,2)}.

L'ensemble des triplets définis par la relation est le suivant:

{(30,1800,3),(10,1000,3),(50,1750,1),(40,700,1),(20,1800,1),(10,1000,2)}.

Nous avons alors écrit un petit programme PROLOG qui permettait de créer un fichier catalogue, de le mettre dans une table et qui appelait notre procédure `meilleur_prix/4`. A la suite de quoi, nous avons posé des questions concernant chacun des usages spécifiés.

Nous nous limiterons ici à un bref échantillon:

1)	query(10, _PV,_F).	Réponse :	(10,1000,3) (10,1000,2)
2)	query(_P,_PV,2).		(10,1000,2)
3)	query(10,_PV,3).		(10,100,3)
4)	query(10,1000,2).		yes
5)	query(_P,jupe,4).		no
6)	query(_PV,_PV,3).		no
7)	query([a,b],[c,d,e],benoît).		no
8)	query([a,b],_PV,4).		no

Nous constatons que, malgré les contraintes de type allégées, la réponse renvoyée est correcte quel que soit le type des paramètres en entrée.
De plus, la procédure est véritablement multi-directionnelle puisqu'en plus des quatre usages présentés dans

le tableau des tests, à la question query(_P,_PV,_F) qui concerne le cinquième usage de la spécification, la réponse fournie fut correcte; les six triplets définis par la relation furent affichés.
La procédure étant véritablement multi-directionnelle, était-elle efficace? Afin d'étudier ce point, le catalogue fut multiplié par cinq. Malheureusement, bien que correcte, la procédure s'avéra peu rapide. Afin d'optimiser le temps de réponse, la description logique fut récrite. La procédure correspondante conserva sa multi-directionnalité et s'avéra très efficace. En voici le code:

```
meilleur_prix(_P,_PV,_F,[él_cata(_P,_PV,_F)|_CAT_FT],):-
                        test_mp(_P,_PV,_CAT_FT).

meilleur_prix(_P,_PV,_F,[él_cata(_P,_PV1,_F1)|_CAT_FT]):-
                        meilleur_prix(_P,_PV,_F,_CAT_FT),
                        meilleur_prix_1(_P,_P1,_PV,_PV1,_F).

meilleur_prix_1(_P,_P1,_PV,_PV1,_F):-
                        geq(_PV1,_PV).

meilleur_prix_1(_P,_P1,_PV,_PV1,_F):-
                        not(_P1=_P).

test_mp(_P,_PV,[]).

test_mp(_P,_PV,[cata_e(_P1,_PV1,_F1)|_CAT_FT]):-
                        test_mp(_P,_PV,_CAT_FT),
                        test_mp(_P,_P1,_PV,_PV1).

test_mp(_P,_P1,_PV,_PV1):-
                        geq(_PV1,_PV).

test_mp(_P,_P1,_PV,_PV1):-
                        not(_P1=_P).

geq(_PV1,_PV):-         _PV1>=_PV.
```

5.3.2.3. Commentaires.

Comme pour les procédures traditionnelles PROLOG, nous avons pu constater qu'un élargissement du domaine d'application de la procédure par un relâchement des contraintes de type (le type `i_a` est remplacé par `any`) permettait d'obtenir plus de permutations correctes et/ou des permutations communes aux différents usages de la procédure.

5.4. CONCLUSION GENERALE.

Pour les procédures `efface/3`, `reverse/2`, `append/3` et `meilleur_prix/4`, lorsque nous avons

supprimé les contraintes de type en entrée et donc élargi le domaine d'application de la procédure, nous avons obtenu plus de permutations correctes et même, pour `efface/3`,, `append/3` et `meilleur_prix/4`, des permutations communes aux différents usages et obtenu, par le fait même des procédures véritablement multi-directionnelles. Dans tous ces cas, `anytype` est remplacé avantageusement par `anyterm`. Le type checking implicite exprimé par `anytype` n'apparaît pas comme étant strictement nécessaire.Cependant, il existe des exemples où l'inverse se produit. Prenons l'exemple suivant:

procédure `p (X)`

Type : `X : list`

Conditions d'application:
in `(anylist)::` **out** `(anylist)`

Son code est le suivant: p(_X).

Pour la directionnalité ci-dessus, il existe une permutation. Si nous lui donnons la directionnalité suivante: **in** `(any)::` **out** `(anylist)`, il n'existe pas de permutation correcte.

En fait, le `anytype` peut être remplacé par `anyterm` si, dans le cas de base (sans récursion), il y a un type checking implicite (par l'unification).
Par exemple, dans:

```
append (L1,L2,L3) ←   =(L1,[]),
                      =(L2,L3).
```

lorsque `L2` est une `groundlist`, `L3` peut être `any` et non `anylist`.

En fait, on peut dire, en général, que remplacer `anytype` par `anyterm` a des effets différents selon que l'on examine la clause de base ou la clause rêcursive. Pour la clause récursive, le nombre de permutations correctes augmente car l'appel récursif est moins contraignant. Pour la clause de base, le nombre de permutations correctes diminue à moins d'un type checking implicite.

6. EXTENSIONS POSSIBLES.

Nulle chose n'étant parfaite, il serait exagéré de dire que l'analyseur répond à toutes les attentes.
D'un point de vue théorique (étude de clauses), nous n'avons qu'à nous louer de son utilisation. Chaque réponse fournie s'est avérée correcte. Aucune mauvaise réponse (permutation erronée) ne nous a été fournie. Certaines permutations que nous croyions correctes se sont avérées fausses et par une étude détaillée, nous nous sommes rendus compte de nos erreurs. Cependant, comme expliqué dans le chapitre 5, un soupçon d'imprécision peut apparaître dans certains cas. Les réponses sont également fournies dans un laps de temps très acceptable.
D'un point de vue pratique, certaines choses peuvent encore être améliorées. Bien que relativement convivial et facile d'utilisation, il serait bon, lors de l'utilisation du module de dérivation (le plus utilisé dans nos tests), de pouvoir changer les directionnalités et les behaviours à tester sans devoir, à chaque fois, sortir du module de dérivation.Que de temps et de manipulations gaspillés. Il serait également utile, lors de la création de spécifications de procédures, de pouvoir lister d'autres spécifications afin de connaître le formalisme. L'édition de descriptions logiques via l'utilisation du "squelette" n'est pas toujours aisée et il est souvent plus facile et rapide de l'éditer directement en faisant référence à une autre.Le prototypage s'est avéré assez pratique, lors de l'étude de `meilleur_prix/4` notamment.
Une extension actuelle est parfaite et bien pratique. C'est la création automatique des primitives lors de la création de nouveaux types. L'utilisateur est libéré d'une tâche relativement délicate et fastidieuse. Il existe toutefois la possibilité de les modifier manuellement.
Si les temps de réponse fournis par le module de dérivation sont acceptables, le temps de chargement de l'analyseur complet est assez long.

7. CONCLUSION.

Lors de l'utilisation de l'analyseur, de nombreux tests ont été effectués et l'analyseur a bien répondu à l'attente. Toutes les permutations de clauses proposées étaient correctes et aucune ne manquait. Nous avons pu remarquer, mais ce n'est pas vrai dans tous les cas, que le type `anytype` en entrée pouvait être remplacé avantageusement par `anyterm`. Nous obtenions plus de permutations correctes et souvent, des permutations correctes communes et ainsi, des procédures véritablement multi-directionnelles.
PROLOG, au début de ce mémoire, était un parfait inconnu. Ce mémoire m'a permis, grâce à l'analyseur, de le connaître ainsi que ses mécanismes (récursivité, backtracking, ...). Ce fut possible grâce à l'utilisation d'un outil performant permettant d'analyser les paramètres de la procédure à chaque point d'exécution de celle-ci. Pour les besoins des tests, il a fallu "manuellement" exécuter ces calculs et les comparer à ceux de l'analyseur. Ce travail de calcul de substitutions s'il est fastidieux est aussi très instructif et riche en enseignement. Il faut reconnaître que l'analyseur remplace avantageusement ce travail.
Il me semble également que cet outil peut être utile pour un "professionnel" PROLOG. En effet, l'analyseur dérive rapidement les permutations correctes d'une procédure PROLOG.Il permet le cas échéant d'analyser en détail pourquoi une pemutation qu'on lui propose est refusée. Programmer en PROLOG n'est pas programmer purement en logique. L'ordre des clauses est important et de plus, certains built-in PROLOG ont des exigences bien particulières quant aux types et modes des paramètres.L'analyseur permet de vérifier rapidement si ces exigences sont satisfaites.
Une autre facilité importante offerte par l'analyseur est la non nécessité de connaître le code des sous-procédures de la procédure à analyser. Seule leur directionnalité est nécessaire (types et modes des paramètres en entrée et en sortie).

Bibliographie

[1] Deville, Y., Logic Programming: Systematic Program Development, Addison-Wesley,1990.

[2] Shapiro, E., and Sterling, L., The Art of Prolog: Advanced Programming Techniques, The MIT Press, 1986.

[3] Clocksin, W.F. and Mellish, C.S. Programming in Prolog. Berlin: Springer-Verlag, 1984.

[4] K.L.Clark, Negation as Failure. In Gallaire and Minker,editors, Logic and Databases. Plenum Press, New York, 1978.

[5] R.A.Kowalski, Logic for Problem Solving. Amsterdam: North-Holland, 1979.

[6] J.W.Lloyd, Foundations of Logic Programming. Springer Series: Symbolic Computation-Artificial Intelligence. Springer-Verlag, second, extended edition, 1987.

[7] P.De Boeck and B.Le Charlier, Somes Lessons Drawn from Using Static Type Analysis for Ensuring the Correctness of Logic Program,University of Namur, 1992.

[8] P.De Boeck and B.Le Charlier, Using Static Type Analysis for Constructing Correct Prolog Procedures: Somes Lessons Learned, University of Namur, 1992.

[9] P.De Boeck, Static Type Analysis of Prolog Procedures for Ensuring Correctness: A User-minded Report,University of Namur, 1990.

[10] P.De Boeck, Deriving Prolog Procedure from Correct Logic Description: A General Survey,University of Namur, 1989.

[11] J.Henrard and B.Le Charlier, FOLON: An Environment for Declarative Construction of Logic Program, in Proc. of PLILP'92, Leuven, August 1992.

[12] P.De Boeck and B.Le Charlier, Static Type Analysis of Prolog Procedures for Ensuring Correctness, in Proceedings of Programming Language Implementation and Logic Programming (PLILP'90), volume 456 of Lecture Notes in Computer Science, pages 222-237, Linköping, Sweden, August 1990, Springer-Verlag.

Annexes.

Les annexes reprennent une partie des tests effectués lors de l'étude critique de l'analyseur ainsi que le code des procédures implémentées pendant l'étude de l'application de "Gestion de Commandes Clients et Fournisseurs". Seuls les exemples les plus significatifs de la façon de travailler de l'analyseur apparaissent dans ces annexes.

Dans l'ordre nous trouverons :

1. Tests effectués sur `efface/3`.
2. Tests effectués sur `member/2`.
3. Tests effectués sur `length/2`.
4. Tests effectués sur `reverse/2`.
5. Tests effectués sur `append/3`.
6. Code PROLOG de la procédure et tests effectués sur `meilleur_prix/4`.

1. TESTS EFFECTUES SUR EFFACE/3.

```
**********************************************
***Results of the syntactical transformation***
**********************************************
```

Clause 1:
****

```
  effbgb(X,L,LEff):-
effbgb(X,T,TEff),
=(LEff,[H|TEff]),
not(=(H,X)),
=(L,[H|T]).
```

Clause 2:
****

```
  effbgb(X,L,LEff):-
=(H,X),
=(LEff,T),
=(L,[H|T]).
```

Analysis of clause 1:
****

```
  effbgb(X,L,LEff):-
effbgb(X,T,TEff),
=(LEff,[H|TEff]),
not(=(H,X)),
=(L,[H|T]).
```

Directionality 1:
****

```
   X/$ground(1),
   L/$groundlist(2),
```

```
   LEff/$anylist(3)
     ::
   X/$ground(1),
   L/$groundlist(2),
   LEff/$groundlist(3).
```

****

```
Permutation 1:
**************

  effbgb(X,L,LEff):-
=(L,[H|T]),
not(=(H,X)),
effbgb(X,T,TEff),
=(LEff,[H|TEff]).


Permutation 2:
**************

  effbgb(X,L,LEff):-
=(L,[H|T]),
effbgb(X,T,TEff),
=(LEff,[H|TEff]),
not(=(H,X)).


Permutation 3:
**************

  effbgb(X,L,LEff):-
=(L,[H|T]),
effbgb(X,T,TEff),
not(=(H,X)),
=(LEff,[H|TEff]).
```

(No more permutations for that clause!)

```
     Analysis of clause 2:
     *********************

  effbgb(X,L,LEff):-
=(H,X),
=(LEff,T),
=(L,[H|T]).


Directionality 1:
*****************

   X/$ground(1),
   L/$groundlist(2),
   LEff/$anylist(3)
     ::
   X/$ground(1),
   L/$groundlist(2),
   LEff/$groundlist(3).

             ************************************


Permutation 1:
**************

  effbgb(X,L,LEff):-
=(H,X),
=(LEff,T),
=(L,[H|T]).


Permutation 2:
**************

  effbgb(X,L,LEff):-
=(H,X),
=(L,[H|T]),
```

```
=(LEff,T).


Permutation 3:
**************

  effbgb(X,L,LEff):-
=(LEff,T),
=(L,[H|T]),
=(H,X).


Permutation 4:
**************

  effbgb(X,L,LEff):-
=(LEff,T),
=(H,X),
=(L,[H|T]).


Permutation 5:
**************

  effbgb(X,L,LEff):-
=(L,[H|T]),
=(LEff,T),
=(H,X).


Permutation 6:
**************

  effbgb(X,L,LEff):-
=(L,[H|T]),
=(H,X),
=(LEff,T).
```

(No more permutations for that clause!)

****
End of the analysis
****

```
*********************************************
***Results of the syntactical transformation***
*********************************************
```

Clause 1:
****

```
  effbgb(X,L,LEff):-
effbgb(X,T,TEff),
=(LEff,[H|TEff]),
not(=(H,X)),
=(L,[H|T]).
```

Clause 2:
****

```
  effbgb(X,L,LEff):-
=(H,X),
=(LEff,T),
=(L,[H|T]).
```

Analysis of clause 1:
****

```
  effbgb(X,L,LEff):-
effbgb(X,T,TEff),
=(LEff,[H|TEff]),
not(=(H,X)),
=(L,[H|T]).
```

Directionality 1:
****

```
   X/$ground(1),
   L/$ground(2),
   LEff/$any(3)
```

```
    ::
   X/$ground(1),
   L/$ground(2),
   LEff/$ground(3).
```

****

```
Permutation 1:
**************

  effbgb(X,L,LEff):-
=(LEff,[H|TEff]),
=(L,[H|T]),
effbgb(X,T,TEff),
not(=(H,X)).


Permutation 2:
**************

  effbgb(X,L,LEff):-
=(LEff,[H|TEff]),
=(L,[H|T]),
not(=(H,X)),
effbgb(X,T,TEff).


Permutation 3:
**************

  effbgb(X,L,LEff):-
=(L,[H|T]),
not(=(H,X)),
=(LEff,[H|TEff]),
effbgb(X,T,TEff).


Permutation 4:
**************

  effbgb(X,L,LEff):-
=(L,[H|T]),
not(=(H,X)),
effbgb(X,T,TEff),
```

```
=(LEff,[H|TEff]).


Permutation 5:
**************

  effbgb(X,L,LEff):-
=(L,[H|T]),
=(LEff,[H|TEff]),
effbgb(X,T,TEff),
not(=(H,X)).


Permutation 6:
**************

  effbgb(X,L,LEff):-
=(L,[H|T]),
=(LEff,[H|TEff]),
not(=(H,X)),
effbgb(X,T,TEff).


Permutation 7:
**************

  effbgb(X,L,LEff):-
=(L,[H|T]),
effbgb(X,T,TEff),
=(LEff,[H|TEff]),
not(=(H,X)).


Permutation 8:
**************

  effbgb(X,L,LEff):-
=(L,[H|T]),
effbgb(X,T,TEff),
not(=(H,X)),
=(LEff,[H|TEff]).


        (No more permutations for that clause!)
```

```
     Analysis of clause 2:
     *********************

  effbgb(X,L,LEff):-
=(H,X),
=(LEff,T),
=(L,[H|T]).


Directionality 1:
*****************

   X/$ground(1),
   L/$ground(2),
   LEff/$any(3)
     ::
   X/$ground(1),
   L/$ground(2),
   LEff/$ground(3).

            ************************************


Permutation 1:
**************

  effbgb(X,L,LEff):-
=(H,X),
=(LEff,T),
=(L,[H|T]).


Permutation 2:
**************
```

```
  effbgb(X,L,LEff):-
=(H,X),
=(L,[H|T]),
=(LEff,T).


Permutation 3:
**************

  effbgb(X,L,LEff):-
=(LEff,T),
=(L,[H|T]),
=(H,X).


Permutation 4:
**************

  effbgb(X,L,LEff):-
=(LEff,T),
=(H,X),
=(L,[H|T]).


Permutation 5:
**************

  effbgb(X,L,LEff):-
=(L,[H|T]),
=(LEff,T),
=(H,X).


Permutation 6:
**************

  effbgb(X,L,LEff):-
=(L,[H|T]),
=(H,X),
=(LEff,T).
```

(No more permutations for that clause!)

```
**************************************************
          ***End of the analysis***
**************************************************
```

```
*************************************************
***Results of the syntactical transformation***
*************************************************
```

```
     Clause 1:
     *********

  effbgb(X,L,LEff):-
effbgb(X,T,TEff),
=(LEff,[H|TEff]),
not(=(H,X)),
=(L,[H|T]).
```

```
     Clause 2:
     *********

  effbgb(X,L,LEff):-
=(H,X),
=(LEff,T),
=(L,[H|T]).
```

```
     Analysis of clause 1:
     *********************

  effbgb(X,L,LEff):-
effbgb(X,T,TEff),
=(LEff,[H|TEff]),
not(=(H,X)),
=(L,[H|T]).
```

```
Directionality 1:
*****************

   X/$ground(1),
   L/$groundlist(2),
   LEff/$var(3)
```

```
    ::
   X/$ground(1),
   L/$groundlist(2),
   LEff/$groundlist(3).

            ************************************
```

```
Permutation 1:
**************

  effbgb(X,L,LEff):-
=(LEff,[H|TEff]),
=(L,[H|T]),
effbgb(X,T,TEff),
not(=(H,X)).


Permutation 2:
**************

  effbgb(X,L,LEff):-
=(LEff,[H|TEff]),
=(L,[H|T]),
not(=(H,X)),
effbgb(X,T,TEff).


Permutation 3:
**************

  effbgb(X,L,LEff):-
=(L,[H|T]),
not(=(H,X)),
=(LEff,[H|TEff]),
effbgb(X,T,TEff).


Permutation 4:
**************

  effbgb(X,L,LEff):-
=(L,[H|T]),
not(=(H,X)),
effbgb(X,T,TEff),
```

```
=(LEff,[H|TEff]).


Permutation 5:
**************

  effbgb(X,L,LEff):-
=(L,[H|T]),
=(LEff,[H|TEff]),
effbgb(X,T,TEff),
not(=(H,X)).


Permutation 6:
**************

  effbgb(X,L,LEff):-
=(L,[H|T]),
=(LEff,[H|TEff]),
not(=(H,X)),
effbgb(X,T,TEff).


Permutation 7:
**************

  effbgb(X,L,LEff):-
=(L,[H|T]),
effbgb(X,T,TEff),
=(LEff,[H|TEff]),
not(=(H,X)).


Permutation 8:
**************

  effbgb(X,L,LEff):-
=(L,[H|T]),
effbgb(X,T,TEff),
not(=(H,X)),
=(LEff,[H|TEff]).


        (No more permutations for that clause!)
```

```
     Analysis of clause 2:
     *********************

  effbgb(X,L,LEff):-
=(H,X),
=(LEff,T),
=(L,[H|T]).



Directionality 1:
*****************

   X/$ground(1),
   L/$groundlist(2),
   LEff/$var(3)
     ::
   X/$ground(1),
   L/$groundlist(2),
   LEff/$groundlist(3).

            ************************************



Permutation 1:
**************

  effbgb(X,L,LEff):-
=(H,X),
=(LEff,T),
=(L,[H|T]).


Permutation 2:
**************
```

```
  effbgb(X,L,LEff):-
=(H,X),
=(L,[H|T]),
=(LEff,T).


Permutation 3:
**************

  effbgb(X,L,LEff):-
=(LEff,T),
=(L,[H|T]),
=(H,X).


Permutation 4:
**************

  effbgb(X,L,LEff):-
=(LEff,T),
=(H,X),
=(L,[H|T]).


Permutation 5:
**************

  effbgb(X,L,LEff):-
=(L,[H|T]),
=(LEff,T),
=(H,X).


Permutation 6:
**************

  effbgb(X,L,LEff):-
=(L,[H|T]),
=(H,X),
=(LEff,T).


        (No more permutations for that clause!)
```

```
**************************************************
           ***End of the analysis***
**************************************************
```

```
**********************************************
***Results of the syntactical transformation***
**********************************************
```

```
     Clause 1:
     *********
```

```
  effbgb(X,L,LEff):-
effbgb(X,T,TEff),
=(LEff,[H|TEff]),
not(=(H,X)),
=(L,[H|T]).
```

```
     Clause 2:
     *********
```

```
  effbgb(X,L,LEff):-
=(H,X),
=(LEff,T),
=(L,[H|T]).
```

```
     Analysis of clause 1:
     *********************
```

```
  effbgb(X,L,LEff):-
effbgb(X,T,TEff),
=(LEff,[H|TEff]),
not(=(H,X)),
=(L,[H|T]).
```

```
Directionality 1:
*****************

   X/$ground(1),
   L/$groundlist(2),
```

```
   LEff/[$any(3)|$anylist(4)]
     ::
   X/$ground(1),
   L/$groundlist(2),
   LEff/$groundlist(3).
```

****

(There is no permutations for that clause!)

Analysis of clause 2:
****

```
  effbgb(X,L,LEff):-
=(H,X),
=(LEff,T),
=(L,[H|T]).
```

Directionality 1:
****

```
   X/$ground(1),
   L/$groundlist(2),
   LEff/[$any(3)|$anylist(4)]
     ::
   X/$ground(1),
   L/$groundlist(2),
   LEff/$groundlist(3).
```

****

```
Permutation 1:
**************

  effbgb(X,L,LEff):-
=(H,X),
=(LEff,T),
=(L,[H|T]).


Permutation 2:
**************

  effbgb(X,L,LEff):-
=(H,X),
=(L,[H|T]),
=(LEff,T).


Permutation 3:
**************

  effbgb(X,L,LEff):-
=(LEff,T),
=(L,[H|T]),
=(H,X).


Permutation 4:
**************

  effbgb(X,L,LEff):-
=(LEff,T),
=(H,X),
=(L,[H|T]).


Permutation 5:
**************

  effbgb(X,L,LEff):-
=(L,[H|T]),
=(LEff,T),
=(H,X).
```

```
Permutation 6:
**************

  effbgb(X,L,LEff):-
=(L,[H|T]),
=(H,X),
=(LEff,T).
```

(No more permutations for that clause!)

```
**************************************************
           ***End of the analysis***
*****************
```

```
*************************************************
***Results of the syntactical transformation***
*************************************************
```

```
     Clause 1:
     *********

  effbgb(X,L,LEff):-
effbgb(X,T,TEff),
=(LEff,[H|TEff]),
not(=(H,X)),
=(L,[H|T]).
```

```
     Clause 2:
     *********

  effbgb(X,L,LEff):-
=(H,X),
=(LEff,T),
=(L,[H|T]).
```

```
     Analysis of clause 1:
     *********************

  effbgb(X,L,LEff):-
effbgb(X,T,TEff),
=(LEff,[H|TEff]),
not(=(H,X)),
=(L,[H|T]).
```

```
Directionality 1:
*****************

   X/$ground(1),
   L/$groundlist(2),
   LEff/[$any(3)|$anylist(4)],
```

```
   ~($any(3),$anylist(4))
     ::
   X/$ground(1),
   L/$groundlist(2),
   LEff/$groundlist(3).
```

****

(There is no permutations for that clause!)

Analysis of clause 2:
****

```
  effbgb(X,L,LEff):-
=(H,X),
=(LEff,T),
=(L,[H|T]).
```

Directionality 1:
****

```
   X/$ground(1),
   L/$groundlist(2),
   LEff/[$any(3)|$anylist(4)],
   ~($any(3),$anylist(4))
     ::
   X/$ground(1),
   L/$groundlist(2),
   LEff/$groundlist(3).
```

****

```
Permutation 1:
**************

  effbgb(X,L,LEff):-
=(H,X),
=(LEff,T),
=(L,[H|T]).


Permutation 2:
**************

  effbgb(X,L,LEff):-
=(H,X),
=(L,[H|T]),
=(LEff,T).


Permutation 3:
**************

  effbgb(X,L,LEff):-
=(LEff,T),
=(L,[H|T]),
=(H,X).


Permutation 4:
**************

  effbgb(X,L,LEff):-
=(LEff,T),
=(H,X),
=(L,[H|T]).


Permutation 5:
**************

  effbgb(X,L,LEff):-
=(L,[H|T]),
=(LEff,T),
=(H,X).
```

```
Permutation 6:
**************

  effbgb(X,L,LEff):-
=(L,[H|T]),
=(H,X),
=(LEff,T).
```

(No more permutations for that clause!)

****

***End

```
*********************************************
***Results of the syntactical transformation***
*********************************************
```

```
     Clause 1:
     *********
```

```
  effbgb(X,L,LEff):-
effbgb(X,T,TEff),
=(LEff,[H|TEff]),
not(=(H,X)),
=(L,[H|T]).
```

```
     Clause 2:
     *********
```

```
  effbgb(X,L,LEff):-
=(H,X),
=(LEff,T),
=(L,[H|T]).
```

```
     Analysis of clause 1:
     *********************
```

```
  effbgb(X,L,LEff):-
effbgb(X,T,TEff),
=(LEff,[H|TEff]),
not(=(H,X)),
=(L,[H|T]).
```

```
Directionality 1:
*****************

   X/$ground(1),
   L/$anylist(2),
```

```
   LEff/$groundlist(3)
     ::
   X/$ground(1),
   L/$groundlist(2),
   LEff/$groundlist(3).
```

****

```
Permutation 1:
**************

  effbgb(X,L,LEff):-
=(LEff,[H|TEff]),
not(=(H,X)),
effbgb(X,T,TEff),
=(L,[H|T]).


Permutation 2:
**************

  effbgb(X,L,LEff):-
=(LEff,[H|TEff]),
effbgb(X,T,TEff),
=(L,[H|T]),
not(=(H,X)).


Permutation 3:
**************

  effbgb(X,L,LEff):-
=(LEff,[H|TEff]),
effbgb(X,T,TEff),
not(=(H,X)),
=(L,[H|T]).


        (No more permutations for that clause!)
```

```
     Analysis of clause 2:
     *********************

  effbgb(X,L,LEff):-
=(H,X),
=(LEff,T),
=(L,[H|T]).


Directionality 1:
*****************

   X/$ground(1),
   L/$anylist(2),
   LEff/$groundlist(3)
     ::
   X/$ground(1),
   L/$groundlist(2),
   LEff/$groundlist(3).

            ************************************


Permutation 1:
**************

  effbgb(X,L,LEff):-
=(H,X),
=(LEff,T),
=(L,[H|T]).


Permutation 2:
**************

  effbgb(X,L,LEff):-
=(H,X),
=(L,[H|T]),
```

```
=(LEff,T).


Permutation 3:
**************

  effbgb(X,L,LEff):-
=(LEff,T),
=(L,[H|T]),
=(H,X).


Permutation 4:
**************

  effbgb(X,L,LEff):-
=(LEff,T),
=(H,X),
=(L,[H|T]).


Permutation 5:
**************

  effbgb(X,L,LEff):-
=(L,[H|T]),
=(LEff,T),
=(H,X).


Permutation 6:
**************

  effbgb(X,L,LEff):-
=(L,[H|T]),
=(H,X),
=(LEff,T).
```

(No more permutations for that clause!)

****
End of the analysis
****

```
**********************************************
***Results of the syntactical transformation***
**********************************************
```

```
     Clause 1:
     *********
```

```
  effbgb(X,L,LEff):-
effbgb(X,T,TEff),
=(LEff,[H|TEff]),
not(=(H,X)),
=(L,[H|T]).
```

```
     Clause 2:
     *********
```

```
  effbgb(X,L,LEff):-
=(H,X),
=(LEff,T),
=(L,[H|T]).
```

```
     Analysis of clause 1:
     *********************
```

```
  effbgb(X,L,LEff):-
effbgb(X,T,TEff),
=(LEff,[H|TEff]),
not(=(H,X)),
=(L,[H|T]).
```

```
Directionality 1:
*****************

   X/$ground(1),
   L/$any(2),
```

```
   LEff/$groundlist(3)
     ::
   X/$ground(1),
   L/$groundlist(2),
   LEff/$groundlist(3).
```

****

Permutation 1:
****

```
  effbgb(X,L,LEff):-
=(LEff,[H|TEff]),
not(=(H,X)),
=(L,[H|T]),
effbgb(X,T,TEff).
```

Permutation 2:
****

```
  effbgb(X,L,LEff):-
=(LEff,[H|TEff]),
not(=(H,X)),
effbgb(X,T,TEff),
=(L,[H|T]).
```

Permutation 3:
****

```
  effbgb(X,L,LEff):-
=(LEff,[H|TEff]),
=(L,[H|T]),
effbgb(X,T,TEff),
not(=(H,X)).
```

Permutation 4:
****

```
  effbgb(X,L,LEff):-
=(LEff,[H|TEff]),
=(L,[H|T]),
```

```
not(=(H,X)),
effbgb(X,T,TEff).


Permutation 5:
**************

  effbgb(X,L,LEff):-
=(LEff,[H|TEff]),
effbgb(X,T,TEff),
=(L,[H|T]),
not(=(H,X)).


Permutation 6:
**************

  effbgb(X,L,LEff):-
=(LEff,[H|TEff]),
effbgb(X,T,TEff),
not(=(H,X)),
=(L,[H|T]).


Permutation 7:
**************

  effbgb(X,L,LEff):-
=(L,[H|T]),
=(LEff,[H|TEff]),
effbgb(X,T,TEff),
not(=(H,X)).


Permutation 8:
**************

  effbgb(X,L,LEff):-
=(L,[H|T]),
=(LEff,[H|TEff]),
not(=(H,X)),
effbgb(X,T,TEff).
```

(No more permutations for that clause!)

```
     Analysis of clause 2:
     *********************

  effbgb(X,L,LEff):-
=(H,X),
=(LEff,T),
=(L,[H|T]).



Directionality 1:
*****************

   X/$ground(1),
   L/$any(2),
   LEff/$groundlist(3)
     ::
   X/$ground(1),
   L/$groundlist(2),
   LEff/$groundlist(3).

            ************************************



Permutation 1:
**************

  effbgb(X,L,LEff):-
=(H,X),
=(LEff,T),
=(L,[H|T]).


Permutation 2:
**************
```

```
  effbgb(X,L,LEff):-
=(H,X),
=(L,[H|T]),
=(LEff,T).


Permutation 3:
**************

  effbgb(X,L,LEff):-
=(LEff,T),
=(L,[H|T]),
=(H,X).


Permutation 4:
**************

  effbgb(X,L,LEff):-
=(LEff,T),
=(H,X),
=(L,[H|T]).


Permutation 5:
**************

  effbgb(X,L,LEff):-
=(L,[H|T]),
=(LEff,T),
=(H,X).


Permutation 6:
**************

  effbgb(X,L,LEff):-
=(L,[H|T]),
=(H,X),
=(LEff,T).
```

(No more permutations for that clause!)

```
**************************************************
          ***End of the analysis***
**************************************************
```

```
*************************************************
***Results of the syntactical transformation***
*************************************************
```

Clause 1:
****

```
  effbgb(X,L,LEff):-
effbgb(X,T,TEff),
=(LEff,[H|TEff]),
not(=(H,X)),
=(L,[H|T]).
```

Clause 2:
****

```
  effbgb(X,L,LEff):-
=(H,X),
=(LEff,T),
=(L,[H|T]).
```

Analysis of clause 1:
****

```
  effbgb(X,L,LEff):-
effbgb(X,T,TEff),
=(LEff,[H|TEff]),
not(=(H,X)),
=(L,[H|T]).
```

```
Directionality 1:
*****************

   X/$any(1),
   L/$groundlist(2),
   LEff/$anylist(3)
     ::
```

```
   X/$ground(1),
   L/$groundlist(2),
   LEff/$groundlist(3).
```

****

```
Permutation 1:
**************

  effbgb(X,L,LEff):-
=(L,[H|T]),
effbgb(X,T,TEff),
=(LEff,[H|TEff]),
not(=(H,X)).


Permutation 2:
**************

  effbgb(X,L,LEff):-
=(L,[H|T]),
effbgb(X,T,TEff),
not(=(H,X)),
=(LEff,[H|TEff]).
```

(No more permutations for that clause!)

Analysis of clause 2:
****

```
  effbgb(X,L,LEff):-
=(H,X),
=(LEff,T),
```

```
=(L,[H|T]).
```

```
Directionality 1:
*****************

   X/$any(1),
   L/$groundlist(2),
   LEff/$anylist(3)
     ::
   X/$ground(1),
   L/$groundlist(2),
   LEff/$groundlist(3).
```

****

```
Permutation 1:
**************

  effbgb(X,L,LEff):-
=(H,X),
=(LEff,T),
=(L,[H|T]).


Permutation 2:
**************

  effbgb(X,L,LEff):-
=(H,X),
=(L,[H|T]),
=(LEff,T).


Permutation 3:
**************

  effbgb(X,L,LEff):-
=(LEff,T),
=(L,[H|T]),
=(H,X).
```

```
Permutation 4:
**************

  effbgb(X,L,LEff):-
=(LEff,T),
=(H,X),
=(L,[H|T]).


Permutation 5:
**************

  effbgb(X,L,LEff):-
=(L,[H|T]),
=(LEff,T),
=(H,X).


Permutation 6:
**************

  effbgb(X,L,LEff):-
=(L,[H|T]),
=(H,X),
=(LEff,T).
```

(No more permutations for that clause!)

```
**************************************************
          ***End of the analysis***
**************************************************
```

```
*********************************************
***Results of the syntactical transformation***
*********************************************
```

Clause 1:
****

```
  effbgb(X,L,LEff):-
effbgb(X,T,TEff),
=(LEff,[H|TEff]),
not(=(H,X)),
=(L,[H|T]).
```

Clause 2:
****

```
  effbgb(X,L,LEff):-
=(H,X),
=(LEff,T),
=(L,[H|T]).
```

Analysis of clause 1:
****

```
  effbgb(X,L,LEff):-
effbgb(X,T,TEff),
=(LEff,[H|TEff]),
not(=(H,X)),
=(L,[H|T]).
```

```
Directionality 1:
*****************

   X/$any(1),
   L/$groundlist(2),
   LEff/$any(3)
     ::
```

```
   X/$ground(1),
   L/$groundlist(2),
   LEff/$groundlist(3).

            ************************************


Permutation 1:
**************

  effbgb(X,L,LEff):-
=(LEff,[H|TEff]),
=(L,[H|T]),
effbgb(X,T,TEff),
not(=(H,X)).


Permutation 2:
**************

  effbgb(X,L,LEff):-
=(L,[H|T]),
=(LEff,[H|TEff]),
effbgb(X,T,TEff),
not(=(H,X)).


Permutation 3:
**************

  effbgb(X,L,LEff):-
=(L,[H|T]),
effbgb(X,T,TEff),
=(LEff,[H|TEff]),
not(=(H,X)).


Permutation 4:
**************

  effbgb(X,L,LEff):-
=(L,[H|T]),
effbgb(X,T,TEff),
not(=(H,X)),
=(LEff,[H|TEff]).
```

(No more permutations for that clause!)

Analysis of clause 2:
****

```
  effbgb(X,L,LEff):-
=(H,X),
=(LEff,T),
=(L,[H|T]).
```

```
Directionality 1:
*****************

   X/$any(1),
   L/$groundlist(2),
   LEff/$any(3)
     ::
   X/$ground(1),
   L/$groundlist(2),
   LEff/$groundlist(3).
```

****

```
Permutation 1:
**************

  effbgb(X,L,LEff):-
=(H,X),
=(LEff,T),
=(L,[H|T]).
```

```
Permutation 2:
**************

  effbgb(X,L,LEff):-
=(H,X),
=(L,[H|T]),
=(LEff,T).


Permutation 3:
**************

  effbgb(X,L,LEff):-
=(LEff,T),
=(L,[H|T]),
=(H,X).


Permutation 4:
**************

  effbgb(X,L,LEff):-
=(LEff,T),
=(H,X),
=(L,[H|T]).


Permutation 5:
**************

  effbgb(X,L,LEff):-
=(L,[H|T]),
=(LEff,T),
=(H,X).


Permutation 6:
**************

  effbgb(X,L,LEff):-
=(L,[H|T]),
=(H,X),
=(LEff,T).
```

(No more permutations for that clause!)

****
End of the analysis
****

```
*********************************************
***Results of the syntactical transformation***
*********************************************
```

```
     Clause 1:
     *********
```

```
  effbgb(X,L,LEff):-
effbgb(X,T,TEff),
=(LEff,[H|TEff]),
not(=(H,X)),
=(L,[H|T]).
```

```
     Clause 2:
     *********
```

```
  effbgb(X,L,LEff):-
=(H,X),
=(LEff,T),
=(L,[H|T]).
```

```
     Analysis of clause 1:
     *********************
```

```
  effbgb(X,L,LEff):-
effbgb(X,T,TEff),
=(LEff,[H|TEff]),
not(=(H,X)),
=(L,[H|T]).
```

```
Directionality 1:
*****************

   X/$ground(1),
   L/$groundlist(2),
   LEff/$groundlist(3)
```

```
    ::
   X/$ground(1),
   L/$groundlist(2),
   LEff/$groundlist(3).
```

****

```
Permutation 1:
**************

  effbgb(X,L,LEff):-
=(LEff,[H|TEff]),
not(=(H,X)),
=(L,[H|T]),
effbgb(X,T,TEff).


Permutation 2:
**************

  effbgb(X,L,LEff):-
=(LEff,[H|TEff]),
=(L,[H|T]),
effbgb(X,T,TEff),
not(=(H,X)).


Permutation 3:
**************

  effbgb(X,L,LEff):-
=(LEff,[H|TEff]),
=(L,[H|T]),
not(=(H,X)),
effbgb(X,T,TEff).


Permutation 4:
**************

  effbgb(X,L,LEff):-
=(L,[H|T]),
not(=(H,X)),
=(LEff,[H|TEff]),
```

```
effbgb(X,T,TEff).


Permutation 5:
**************

  effbgb(X,L,LEff):-
=(L,[H|T]),
=(LEff,[H|TEff]),
effbgb(X,T,TEff),
not(=(H,X)).


Permutation 6:
**************

  effbgb(X,L,LEff):-
=(L,[H|T]),
=(LEff,[H|TEff]),
not(=(H,X)),
effbgb(X,T,TEff).
```

(No more permutations for that clause!)

Analysis of clause 2:
****

```
  effbgb(X,L,LEff):-
=(H,X),
=(LEff,T),
=(L,[H|T]).


Directionality 1:
```

```
*****************

   X/$ground(1),
   L/$groundlist(2),
   LEff/$groundlist(3)
     ::
   X/$ground(1),
   L/$groundlist(2),
   LEff/$groundlist(3).

              ************************************


Permutation 1:
**************

  effbgb(X,L,LEff):-
=(H,X),
=(LEff,T),
=(L,[H|T]).


Permutation 2:
**************

  effbgb(X,L,LEff):-
=(H,X),
=(L,[H|T]),
=(LEff,T).


Permutation 3:
**************

  effbgb(X,L,LEff):-
=(LEff,T),
=(L,[H|T]),
=(H,X).


Permutation 4:
**************

  effbgb(X,L,LEff):-
=(LEff,T),
```

```
=(H,X),
=(L,[H|T]).


Permutation 5:
**************

  effbgb(X,L,LEff):-
=(L,[H|T]),
=(LEff,T),
=(H,X).


Permutation 6:
**************

  effbgb(X,L,LEff):-
=(L,[H|T]),
=(H,X),
=(LEff,T).

        (No more permutations for that clause!)
```

****
End of the analysis
****

```
          ***********************************************
          ***Results of the syntactical transformation***
          ***********************************************

     Clause 1:
     *********

  efface(X,L,LEff):-
        efface(X,T,TEff),
        =(LEff,[H|TEff]),
        not(=(H,X)),
        =(L,[H|T]).

     Clause 2:
     *********

  efface(X,L,LEff):-
        =(H,X),
        =(LEff,T),
        =(L,[H|T]).


     Analysis of clause 1:
     *********************

  efface(X,L,LEff):-
        efface(X,T,TEff),
        =(LEff,[H|TEff]),
        not(=(H,X)),
        =(L,[H|T]).


Common permutation 1:
*********************

  efface(X,L,LEff):-
        =(LEff,[H|TEff]),
        =(L,[H|T]),
        efface(X,T,TEff),
        not(=(H,X)).


Common permutation 2:
*********************

  efface(X,L,LEff):-
        =(L,[H|T]),
        =(LEff,[H|TEff]),
        efface(X,T,TEff),
        not(=(H,X)).

        (No more common permutations for that clause!)


     Analysis of clause 2:
     *********************

  efface(X,L,LEff):-
        =(H,X),
        =(LEff,T),
        =(L,[H|T]).


Common permutation 1:
*********************

  efface(X,L,LEff):-
        =(H,X),
        =(LEff,T),
        =(L,[H|T]).


Common permutation 2:
*********************

  efface(X,L,LEff):-
        =(H,X),
        =(L,[H|T]),
        =(LEff,T).


Common permutation 3:
*********************

  efface(X,L,LEff):-
        =(LEff,T),
        =(L,[H|T]),
        =(H,X).


Common permutation 4:
*********************

  efface(X,L,LEff):-
        =(LEff,T),
        =(H,X),
        =(L,[H|T]).


Common permutation 5:
```

****

```
  efface(X,L,LEff):-
        =(L,[H|T]),
        =(LEff,T),
        =(H,X).
```

Common permutation 6:
****

```
  efface(X,L,LEff):-
        =(L,[H|T]),
        =(H,X),
        =(LEff,T).
```

(No more common permutations for that clause!)

****
End of the analysis
****

2. TESTS EFFECTUES SUR MEMBER/2.

```
          ***********************************************
          ***Results of the syntactical transformation***
          ***********************************************


     Clause 1:
     *********

  member(X,L):-
        not(=(X,H)),
        member(X,T),
        =(L,[H|T]).

     Clause 2:
     *********

  member(X,L):-
        =(X,H),
        =(L,[H|T]).



     Analysis of clause 1:
     *********************

  member(X,L):-
        not(=(X,H)),
        member(X,T),
        =(L,[H|T]).


Directionality 1:
*****************

   X/$any(1),
   L/$groundlist(1)
     ::
   X/$ground(1),
   L/$groundlist(1).

            ***********************************


Permutation 1:
**************

  member(X,L):-
        =(L,[H|T]),
        member(X,T),
        not(=(X,H)).

        (No more permutations for that clause!)



     Analysis of clause 2:
     *********************

  member(X,L):-
        =(X,H),
        =(L,[H|T]).


Directionality 1:
*****************

   X/$any(1),
   L/$groundlist(1)
     ::
   X/$ground(1),
   L/$groundlist(1).

            ***********************************


Permutation 1:
**************

  member(X,L):-
        =(X,H),
        =(L,[H|T]).


Permutation 2:
**************

  member(X,L):-
        =(L,[H|T]),
        =(X,H).

        (No more permutations for that clause!)
```

****
End of the analysis
****

****
Results of the syntactical transformation
****

Clause 1:
****

```
member(X,L):-
      not(=(X,H)),
      member(X,T),
      =(L,[H|T]).
```

Clause 2:
****

```
member(X,L):-
      =(X,H),
      =(L,[H|T]).
```

Analysis of clause 1:
****

```
member(X,L):-
      not(=(X,H)),
      member(X,T),
      =(L,[H|T]).
```

Directionality 1:
****

```
X/$ground(1),
L/$groundlist(1)
  ::
X/$ground(1),
L/$groundlist(1).
```

****

Permutation 1:
****

```
member(X,L):-
      =(L,[H|T]),
      member(X,T),
      not(=(X,H)).
```

Permutation 2:
****

```
member(X,L):-
      =(L,[H|T]),
      not(=(X,H)),
      member(X,T).
```

(No more permutations for that clause!)

Analysis of clause 2:
****

```
member(X,L):-
      =(X,H),
      =(L,[H|T]).
```

Directionality 1:
****

```
X/$ground(1),
L/$groundlist(1)
  ::
X/$ground(1),
L/$groundlist(1).
```

****

Permutation 1:
****

```
member(X,L):-
      =(X,H),
      =(L,[H|T]).
```

Permutation 2:
****

```
member(X,L):-
       =(L,[H|T]),
       =(X,H).

       (No more permutations for that clause!)
```

```
**************************************************
          ***End of the analysis***
**************************************************
```

```
*********************************************
***Results of the syntactical transformation***
*********************************************
```

Clause 1:
****

```
member(X,L):-
      not(=(X,H)),
      member(X,T),
      =(L,[H|T]).
```

Clause 2:
****

```
member(X,L):-
      =(X,H),
      =(L,[H|T]).
```

Analysis of clause 1:
****

```
member(X,L):-
      not(=(X,H)),
      member(X,T),
      =(L,[H|T]).
```

Directionality 1:
****

```
X/$ground(1),
L/$anylist(1)
  ::
X/$ground(1),
L/$anylist(1).
```

****

(There is no permutations for that clause!)

Analysis of clause 2:
****

```
member(X,L):-
      =(X,H),
      =(L,[H|T]).
```

Directionality 1:
****

```
X/$ground(1),
L/$anylist(1)
  ::
X/$ground(1),
L/$anylist(1).
```

****

(There is no permutations for that clause!)

```
*************************************************
          ***End of the analysis***
*************************************************
```

****
Results of the syntactical transformation
****

Clause 1:
****

```
member(X,L):-
      not(=(X,H)),
      member(X,T),
      =(L,[H|T]).
```

Clause 2:
****

```
member(X,L):-
      =(X,H),
      =(L,[H|T]).
```

Analysis of clause 1:
****

```
member(X,L):-
      not(=(X,H)),
      member(X,T),
      =(L,[H|T]).
```

Directionality 1:
****

```
X/$ground(1),
L/$anylist(1)
  ::
X/$ground(1),
L/$groundlist(1).
```

****

(There is no permutations for that clause!)

Analysis of clause 2:
****

```
member(X,L):-
      =(X,H),
      =(L,[H|T]).
```

Directionality 1:
****

```
X/$ground(1),
L/$anylist(1)
  ::
X/$ground(1),
L/$groundlist(1).
```

****

(There is no permutations for that clause!)

****
End of the analysis
****

```
**********************************************
***Results of the syntactical transformation***
**********************************************
```

Clause 1:
****

```
member(X,L):-
      not(=(X,H)),
      member(X,T),
      =(L,[H|T]).
```

Clause 2:
****

```
member(X,L):-
      =(X,H),
      =(L,[H|T]).
```

Analysis of clause 1:
****

```
member(X,L):-
      not(=(X,H)),
      member(X,T),
      =(L,[H|T]).
```

Directionality 1:
****

```
X/$ground(1),
L/$any(1)
  ::
X/$ground(1),
L/$anylist(1).
```

****

(There is no permutations for that clause!)

Analysis of clause 2:
****

```
member(X,L):-
      =(X,H),
      =(L,[H|T]).
```

Directionality 1:
****

```
X/$ground(1),
L/$any(1)
  ::
X/$ground(1),
L/$anylist(1).
```

****

(There is no permutations for that clause!)

```
**************************************************
          ***End of the analysis***
**************************************************
```

3. TESTS EFFECTUES SUR LENGTH/2.

****
Results of the syntactical transformation
****

Clause 1:
****

```
length(L,N):-
        length(T,N_T),
        is(N,+(N_T,1)),
        =(L,[H|T]).
```

Clause 2:
****

```
length(L,N):-
        =(L,[]),
        =(N,0).
```

Analysis of clause 1:
****

```
length(L,N):-
        length(T,N_T),
        is(N,+(N_T,1)),
        =(L,[H|T]).
```

Directionality 1:
****

```
L/$groundlist(1),
N/$int_a(1)
  ::
L/$groundlist(1),
N/$integer(2).
```

****

Permutation 1:
****

```
length(L,N):-
        =(L,[H|T]),
        length(T,N_T),
        is(N,+(N_T,1)).
```

(No more permutations for that clause!)

Analysis of clause 2:
****

```
length(L,N):-
        =(L,[]),
        =(N,0).
```

Directionality 1:
****

```
L/$groundlist(1),
N/$int_a(1)
  ::
L/$groundlist(1),
N/$integer(2).
```

****

Permutation 1:
****

```
length(L,N):-
        =(L,[]),
        =(N,0).
```

Permutation 2:
****

```
length(L,N):-
        =(N,0),
        =(L,[]).
```

(No more permutations for that clause!)

****
End of the analysis
****

****
Results of the syntactical transformation
****

Clause 1:
****

```
length(L,N):-
      length(T,N_T),
      is(N,+(N_T,1)),
      =(L,[H|T]).
```

Clause 2:
****

```
length(L,N):-
      =(L,[]),
      =(N,0).
```

Analysis of clause 1:
****

```
length(L,N):-
      length(T,N_T),
      is(N,+(N_T,1)),
      =(L,[H|T]).
```

Directionality 1:
****

```
L/$groundlist(1),
N/$any(1)
  ::
L/$groundlist(1),
N/$integer(2).
```

****

(There is no permutations for that clause!)

Analysis of clause 2:
****

```
length(L,N):-
      =(L,[]),
      =(N,0).
```

Directionality 1:
****

```
L/$groundlist(1),
N/$any(1)
  ::
L/$groundlist(1),
N/$integer(2).
```

****

Permutation 1:
****

```
length(L,N):-
      =(L,[]),
      =(N,0).
```

Permutation 2:
****

```
length(L,N):-
      =(N,0),
      =(L,[]).
```

(No more permutations for that clause!)

****

End of the analysis
****

****
Results of the syntactical transformation
****

Clause 1:
****

```
length(L,N):-
      length(T,N_T),
      is(N,+(N_T,1)),
      =(L,[H|T]).
```

Clause 2:
****

```
length(L,N):-
      =(L,[]),
      =(N,0).
```

Analysis of clause 1:
****

```
length(L,N):-
      length(T,N_T),
      is(N,+(N_T,1)),
      =(L,[H|T]).
```

Directionality 1:
****

```
L/$anylist(1),
N/$integer(1)
  ::
L/$anylist(1),
N/$integer(1).
```

****

(There is no permutations for that clause!)

Analysis of clause 2:
****

```
length(L,N):-
      =(L,[]),
      =(N,0).
```

Directionality 1:
****

```
L/$anylist(1),
N/$integer(1)
  ::
L/$anylist(1),
N/$integer(1).
```

****

Permutation 1:
****

```
length(L,N):-
      =(L,[]),
      =(N,0).
```

Permutation 2:
****

```
length(L,N):-
      =(N,0),
      =(L,[]).
```

(No more permutations for that clause!)

****

End of the analysis
****

****
Results of the syntactical transformation
****

Clause 1:
****

```
length(L,N):-
      length(T,N_T),
      is(N,+(N_T,1)),
      =(L,[H|T]).
```

Clause 2:
****

```
length(L,N):-
      =(L,[]),
      =(N,0).
```

Analysis of clause 1:
****

```
length(L,N):-
      length(T,N_T),
      is(N,+(N_T,1)),
      =(L,[H|T]).
```

Directionality 1:
****

```
L/$any(1),
N/$integer(1)
  ::
L/$anylist(1),
N/$integer(1).
```

****

(There is no permutations for that clause!)

Analysis of clause 2:
****

```
length(L,N):-
      =(L,[]),
      =(N,0).
```

Directionality 1:
****

```
L/$any(1),
N/$integer(1)
  ::
L/$anylist(1),
N/$integer(1).
```

****

Permutation 1:
****

```
length(L,N):-
      =(L,[]),
      =(N,0).
```

Permutation 2:
****

```
length(L,N):-
      =(N,0),
      =(L,[]).
```

(No more permutations for that clause!)

****

End of the analysis
****

```
          ***********************************************
          ***Results of the syntactical transformation***
          ***********************************************


     Clause 1:
     *********

  length(L,N):-
        length(T,N_T),
        is(N,+(N_T,1)),
        =(L,[H|T]).

     Clause 2:
     *********

  length(L,N):-
        =(L,[]),
        =(N,0).




     Analysis of clause 1:
     *********************

  length(L,N):-
        length(T,N_T),
        is(N,+(N_T,1)),
        =(L,[H|T]).


Directionality 1:
*****************

   L/$groundlist(1),
   N/$int_a(1)
     ::
   L/$groundlist(1),
   N/$integer(1).

            ************************************


Permutation 1:
**************

  length(L,N):-
        =(L,[H|T]),
        length(T,N_T),
        is(N,+(N_T,1)).

                (No more permutations for that clause!)


Directionality 2:
*****************

   L/$anylist(1),
   N/$integer(1)
     ::
   L/$anylist(1),
   N/$integer(1).

            ************************************


        (There is no permutations for that clause!)



     Analysis of clause 2:
     *********************

  length(L,N):-
        =(L,[]),
        =(N,0).


Directionality 1:
*****************

   L/$groundlist(1),
   N/$int_a(1)
     ::
   L/$groundlist(1),
   N/$integer(1).

            ************************************


Permutation 1:
```

```
**************

  length(L,N):-
        =(L,[]),
        =(N,0).
```

```
Permutation 2:
**************

  length(L,N):-
        =(N,0),
        =(L,[]).

        (No more permutations for that clause!)
```

```
Directionality 2:
*****************

   L/$anylist(1),
   N/$integer(1)
     ::
   L/$anylist(1),
   N/$integer(1).
```

****

```
Permutation 1:
**************

  length(L,N):-
        =(L,[]),
        =(N,0).

Permutation 2:
**************

  length(L,N):-
        =(N,0),
        =(L,[]).

        (No more permutations for that clause!)
```

****
End of the analysis
****

```
          ***********************************************
           ***Results of the syntactical transformation***
           ***********************************************


      Clause 1:
      *********

  length(L,N):-
        length(T,N_T),
        is(N,+(N_T,1)),
        =(L,[H|T]).

      Clause 2:
      *********

  length(L,N):-
        =(L,[]),
        =(N,0).




      Analysis of clause 1:
      *********************

  length(L,N):-
        length(T,N_T),
        is(N,+(N_T,1)),
        =(L,[H|T]).


        (There is no common permutations for that clause!)



      Analysis of clause 2:
      *********************

  length(L,N):-
        =(L,[]),
        =(N,0).


Common permutation 1:
*********************

  length(L,N):-
        =(L,[]),
        =(N,0).


Common permutation 2:
*********************

  length(L,N):-
        =(N,0),
        =(L,[]).


        (No more common permutations for that clause!)



*****************************************************
           ***End of this analysis***
*****************************************************
```

4. TESTS EFFECTUES SUR REVERSE/2.

```
          ********************************************
           ***Results of the syntactical transformation***
           ********************************************
```

```
     Clause 1:
     *********

  reverse(L,LRev):-
        reverse(T,Rem_LRev),
        append(Rem_LRev,[H],LRev),
        =(L,[H|T]).

     Clause 2:
     *********

  reverse(L,LRev):-
        =(L,[]),
        =(LRev,[]).
```

```
     Analysis of clause 1:
     *********************

  reverse(L,LRev):-
        reverse(T,Rem_LRev),
        append(Rem_LRev,[H],LRev),
        =(L,[H|T]).
```

```
Directionality 1:
*****************

   L/$groundlist(1),
   LRev/$anylist(1)
     ::
   L/$groundlist(1),
   LRev/$groundlist(2).

            ***********************************
```

```
Permutation 1:
**************

  reverse(L,LRev):-
        =(L,[H|T]),
        reverse(T,Rem_LRev),
        append(Rem_LRev,[H],LRev).
```

```
                (No more permutations for that clause!)
```

```
     Analysis of clause 2:
     *********************

  reverse(L,LRev):-
        =(L,[]),
        =(LRev,[]).
```

```
Directionality 1:
*****************

   L/$groundlist(1),
   LRev/$anylist(1)
     ::
   L/$groundlist(1),
   LRev/$groundlist(2).

            ***********************************
```

```
Permutation 1:
**************

  reverse(L,LRev):-
        =(L,[]),
        =(LRev,[]).

Permutation 2:
**************

  reverse(L,LRev):-
        =(LRev,[]),
        =(L,[]).

        (No more permutations for that clause!)
```

****
End of the analysis
****

****
Results of the syntactical transformation
****

Clause 1:
****

```
reverse(L,LRev):-
      reverse(T,Rem_LRev),
      append(Rem_LRev,[H],LRev),
      =(L,[H|T]).
```

Clause 2:
****

```
reverse(L,LRev):-
      =(L,[]),
      =(LRev,[]).
```

Analysis of clause 1:
****

```
reverse(L,LRev):-
      reverse(T,Rem_LRev),
      append(Rem_LRev,[H],LRev),
      =(L,[H|T]).
```

Directionality 1:
****

```
L/$anylist(1),
LRev/$groundlist(1)
  ::
L/$groundlist(1),
LRev/$groundlist(2).
```

****

Permutation 1:
****

```
reverse(L,LRev):-
      append(Rem_LRev,[H],LRev),
      reverse(T,Rem_LRev),
      =(L,[H|T]).
```

(No more permutations for that clause!)

Analysis of clause 2:
****

```
reverse(L,LRev):-
      =(L,[]),
      =(LRev,[]).
```

Directionality 1:
****

```
L/$anylist(1),
LRev/$groundlist(1)
  ::
L/$groundlist(1),
LRev/$groundlist(2).
```

****

Permutation 1:
****

```
reverse(L,LRev):-
      =(L,[]),
      =(LRev,[]).
```

Permutation 2:
****

```
reverse(L,LRev):-
      =(LRev,[]),
      =(L,[]).
```

(No more permutations for that clause!)

****
End of the analysis
****

```
          ***********************************************
         ***Results of the syntactical transformation***
         ***********************************************

    Clause 1:
    *********

reverse(L,LRev):-
      reverse(T,Rem_LRev),
      append(Rem_LRev,[H],LRev),
      =(L,[H|T]).

    Clause 2:
    *********

reverse(L,LRev):-
      =(L,[]),
      =(LRev,[]).




    Analysis of clause 1:
    *********************

reverse(L,LRev):-
      reverse(T,Rem_LRev),
      append(Rem_LRev,[H],LRev),
      =(L,[H|T]).


Directionality 1:
*****************

   L/$groundlist(1),
   LRev/$anylist(2)
     ::
   L/$groundlist(1),
   LRev/$groundlist(2).

           ***********************************


Permutation 1:
**************

reverse(L,LRev):-
      =(L,[H|T]),
      reverse(T,Rem_LRev),
      append(Rem_LRev,[H],LRev).

      (No more permutations for that clause!)


Directionality 2:
*****************

   L/$anylist(1),
   LRev/$groundlist(2)
     ::
   L/$groundlist(1),
   LRev/$groundlist(2).

           ***********************************


Permutation 1:
**************

reverse(L,LRev):-
      append(Rem_LRev,[H],LRev),
      reverse(T,Rem_LRev),
      =(L,[H|T]).

      (No more permutations for that clause!)


    Analysis of clause 2:
    *********************

reverse(L,LRev):-
      =(L,[]),
      =(LRev,[]).


Directionality 1:
*****************

   L/$groundlist(1),
   LRev/$anylist(2)
```

```
     ::
   L/$groundlist(1),
   LRev/$groundlist(2).

            ************************************


Permutation 1:
**************

  reverse(L,LRev):-
        =(L,[]),
        =(LRev,[]).


Permutation 2:
**************

  reverse(L,LRev):-
        =(LRev,[]),
        =(L,[]).

        (No more permutations for that clause!)



Directionality 2:
*****************

   L/$anylist(1),
   LRev/$groundlist(2)
     ::
   L/$groundlist(1),
   LRev/$groundlist(2).

            ************************************


Permutation 1:
**************

  reverse(L,LRev):-
        =(L,[]),
        =(LRev,[]).


Permutation 2:
**************

  reverse(L,LRev):-
        =(LRev,[]),
        =(L,[]).

        (No more permutations for that clause!)


**************************************************
           ***End of the analysis***
**************************************************
```

```
          ***********************************************
           ***Results of the syntactical transformation***
           ***********************************************


     Clause 1:
     *********

  reverse(L,LRev):-
        reverse(T,Rem_LRev),
        append(Rem_LRev,[H],LRev),
        =(L,[H|T]).

     Clause 2:
     *********

  reverse(L,LRev):-
        =(L,[]),
        =(LRev,[]).




     Analysis of clause 1:
     *********************

  reverse(L,LRev):-
        reverse(T,Rem_LRev),
        append(Rem_LRev,[H],LRev),
        =(L,[H|T]).


        (There is no common permutations for that clause!)



     Analysis of clause 2:
     *********************

  reverse(L,LRev):-
        =(L,[]),
        =(LRev,[]).


Common permutation 1:
*********************

  reverse(L,LRev):-
        =(L,[]),
        =(LRev,[]).

Common permutation 2:
*********************

  reverse(L,LRev):-
        =(LRev,[]),
        =(L,[]).


        (No more common permutations for that clause!)



***************************************************
           ***End of the analysis***
***************************************************
```

```
          ***********************************************
          ***Results of the syntactical transformation***
          ***********************************************

     Clause 1:
     *********

  reverse(L,LRev):-
        reverse(T,Rem_LRev),
        append(Rem_LRev,[H],LRev),
        =(L,[H|T]).

     Clause 2:
     *********

  reverse(L,LRev):-
        =(L,[]),
        =(LRev,[]).




     Analysis of clause 1:
     *********************

  reverse(L,LRev):-
        reverse(T,Rem_LRev),
        append(Rem_LRev,[H],LRev),
        =(L,[H|T]).


Directionality 1:
*****************

   L/$groundlist(1),
   LRev/$any(1)
     ::
   L/$groundlist(1),
   LRev/$groundlist(2).

            ************************************


Permutation 1:
**************

  reverse(L,LRev):-
        =(L,[H|T]),
        reverse(T,Rem_LRev),
        append(Rem_LRev,[H],LRev).
```

(No more permutations for that clause!)

```
Directionality 2:
*****************

   L/$any(1),
   LRev/$groundlist(1)
     ::
   L/$groundlist(1),
   LRev/$groundlist(2).

            ************************************


Permutation 1:
**************

  reverse(L,LRev):-
        append(Rem_LRev,[H],LRev),
        =(L,[H|T]),
        reverse(T,Rem_LRev).


Permutation 2:
**************

  reverse(L,LRev):-
        append(Rem_LRev,[H],LRev),
        reverse(T,Rem_LRev),
        =(L,[H|T]).


Permutation 3:
**************

  reverse(L,LRev):-
        =(L,[H|T]),
        append(Rem_LRev,[H],LRev),
        reverse(T,Rem_LRev).
```

(No more permutations for that clause!)

Analysis of clause 2:
****

```
reverse(L,LRev):-
      =(L,[]),
      =(LRev,[]).
```

Directionality 1:
****

```
   L/$groundlist(1),
   LRev/$any(1)
     ::
   L/$groundlist(1),
   LRev/$groundlist(2).
```

****

Permutation 1:
****

```
reverse(L,LRev):-
      =(L,[]),
      =(LRev,[]).
```

Permutation 2:
****

```
reverse(L,LRev):-
      =(LRev,[]),
      =(L,[]).
```

(No more permutations for that clause!)

Directionality 2:
****

```
   L/$any(1),
   LRev/$groundlist(1)
     ::
   L/$groundlist(1),
   LRev/$groundlist(2).
```

****

Permutation 1:
****

```
reverse(L,LRev):-
      =(L,[]),
      =(LRev,[]).
```

Permutation 2:
****

```
reverse(L,LRev):-
      =(LRev,[]),
      =(L,[]).
```

(No more permutations for that clause!)

****
End of the analysis
****

```
          ***********************************************
          ***Results of the syntactical transformation***
          ***********************************************


     Clause 1:
     *********

  reverse(L,LRev):-
        reverse(T,Rem_LRev),
        append(Rem_LRev,[H],LRev),
        =(L,[H|T]).

     Clause 2:
     *********

  reverse(L,LRev):-
        =(L,[]),
        =(LRev,[]).




     Analysis of clause 1:
     *********************

  reverse(L,LRev):-
        reverse(T,Rem_LRev),
        append(Rem_LRev,[H],LRev),
        =(L,[H|T]).


        (There is no common permutations for that clause!)



     Analysis of clause 2:
     *********************

  reverse(L,LRev):-
        =(L,[]),
        =(LRev,[]).


Common permutation 1:
*********************

  reverse(L,LRev):-
        =(L,[]),
        =(LRev,[]).

Common permutation 2:
*********************

  reverse(L,LRev):-
        =(LRev,[]),
        =(L,[]).


        (No more common permutations for that clause!)



***************************************************
           ***End of the analysis***
***************************************************
```

```
          ********************************************
          ***Results of the syntactical transformation***
          ********************************************


     Clause 1:
     *********

  reverse(L,LRev):-
        reverse(T,Rem_LRev),
        append(Rem_LRev,[H],LRev),
        =(L,[H|T]).

     Clause 2:
     *********

  reverse(L,LRev):-
        =(L,[]),
        =(LRev,[]).




     Analysis of clause 1:
     *********************

  reverse(L,LRev):-
        reverse(T,Rem_LRev),
        append(Rem_LRev,[H],LRev),
        =(L,[H|T]).


Directionality 1:
*****************

   L/$groundlist(1),
   LRev/$any(1)
     ::
   L/$groundlist(1),
   LRev/$groundlist(2).

            ***********************************


        (There is no permutations for that clause!)



Directionality 2:
*****************

   L/$any(1),
   LRev/$groundlist(1)
     ::
   L/$groundlist(1),
   LRev/$groundlist(2).

            ***********************************


Permutation 1:
**************

  reverse(L,LRev):-
        append(Rem_LRev,[H],LRev),
        =(L,[H|T]),
        reverse(T,Rem_LRev).


Permutation 2:
**************

  reverse(L,LRev):-
        append(Rem_LRev,[H],LRev),
        reverse(T,Rem_LRev),
        =(L,[H|T]).


Permutation 3:
**************

  reverse(L,LRev):-
        =(L,[H|T]),
        append(Rem_LRev,[H],LRev),
        reverse(T,Rem_LRev).


        (No more permutations for that clause!)




     Analysis of clause 2:
     *********************

  reverse(L,LRev):-
        =(L,[]),
```

```
        =(LRev,[]).


Directionality 1:
*****************

   L/$groundlist(1),
   LRev/$any(1)
     ::
   L/$groundlist(1),
   LRev/$groundlist(2).

            ************************************


Permutation 1:
**************

  reverse(L,LRev):-
        =(L,[]),
        =(LRev,[]).


Permutation 2:
**************

  reverse(L,LRev):-
        =(LRev,[]),
        =(L,[]).


        (No more permutations for that clause!)



Directionality 2:
*****************

   L/$any(1),
   LRev/$groundlist(1)
     ::
   L/$groundlist(1),
   LRev/$groundlist(2).

            ************************************


Permutation 1:
**************

  reverse(L,LRev):-
        =(L,[]),
        =(LRev,[]).


Permutation 2:
**************

  reverse(L,LRev):-
        =(LRev,[]),
        =(L,[]).


        (No more permutations for that clause!)


**************************************************
          ***End of the analysis***
**************************************************
```

5. TESTS EFFECTUES SUR APPEND/3.

```
          ***********************************************
           ***Results of the syntactical transformation***
           ***********************************************


     Clause 1:
     *********

  append(L1,L2,L3):-
        append(T,L2,Rem_L3),
        =(L3,[H|Rem_L3]),
        =(L1,[H|T]).

     Clause 2:
     *********

  append(L1,L2,L3):-
        =(L1,[]),
        =(L2,L3).




     Analysis of clause 1:
     *********************

  append(L1,L2,L3):-
        append(T,L2,Rem_L3),
        =(L3,[H|Rem_L3]),
        =(L1,[H|T]).


Directionality 1:
*****************

   L1/$groundlist(1),
   L2/$groundlist(2),
   L3/$anylist(1)
     ::
   L1/$groundlist(1),
   L2/$groundlist(2),
   L3/$groundlist(3).

            ************************************


Permutation 1:
**************

  append(L1,L2,L3):-
        =(L1,[H|T]),
        append(T,L2,Rem_L3),
        =(L3,[H|Rem_L3]).


        (No more permutations for that clause!)



Directionality 2:
*****************

   L1/$anylist(1),
   L2/$anylist(2),
   L3/$groundlist(1)
     ::
   L1/$groundlist(1),
   L2/$groundlist(2),
   L3/$groundlist(3).

            ************************************


Permutation 1:
**************

  append(L1,L2,L3):-
        =(L3,[H|Rem_L3]),
        append(T,L2,Rem_L3),
        =(L1,[H|T]).


        (No more permutations for that clause!)



     Analysis of clause 2:
     *********************

  append(L1,L2,L3):-
        =(L1,[]),
        =(L2,L3).


Directionality 1:
```

```
*****************

   L1/$groundlist(1),
   L2/$groundlist(2),
   L3/$anylist(1)
     ::
   L1/$groundlist(1),
   L2/$groundlist(2),
   L3/$groundlist(3).

            ************************************
```

```
Permutation 1:
**************

  append(L1,L2,L3):-
        =(L1,[]),
        =(L2,L3).
```

```
Permutation 2:
**************

  append(L1,L2,L3):-
        =(L2,L3),
        =(L1,[]).

        (No more permutations for that clause!)
```

```
Directionality 2:
*****************

   L1/$anylist(1),
   L2/$anylist(2),
   L3/$groundlist(1)
     ::
   L1/$groundlist(1),
   L2/$groundlist(2),
   L3/$groundlist(3).

            ************************************
```

```
Permutation 1:
**************

  append(L1,L2,L3):-
        =(L1,[]),
        =(L2,L3).
```

```
Permutation 2:
**************

  append(L1,L2,L3):-
        =(L2,L3),
        =(L1,[]).

        (No more permutations for that clause!)
```

```
**************************************************
          ***End of the analysis***
**************************************************
```

```
          ***********************************************
           ***Results of the syntactical transformation***
           ***********************************************


     Clause 1:
     *********

  append(L1,L2,L3):-
        append(T,L2,Rem_L3),
        =(L3,[H|Rem_L3]),
        =(L1,[H|T]).

     Clause 2:
     *********

  append(L1,L2,L3):-
        =(L1,[]),
        =(L2,L3).




     Analysis of clause 1:
     *********************

  append(L1,L2,L3):-
        append(T,L2,Rem_L3),
        =(L3,[H|Rem_L3]),
        =(L1,[H|T]).


        (There is no common permutations for that clause!)



     Analysis of clause 2:
     *********************

  append(L1,L2,L3):-
        =(L1,[]),
        =(L2,L3).


Common permutation 1:
*********************

  append(L1,L2,L3):-
        =(L1,[]),
        =(L2,L3).


Common permutation 2:
*********************

  append(L1,L2,L3):-
        =(L2,L3),
        =(L1,[]).


        (No more common permutations for that clause!)



***************************************************
           ***End of the analysis***
***************************************************
```

```
          ***********************************************
          ***Results of the syntactical transformation***
          ***********************************************


     Clause 1:
     *********

  append(L1,L2,L3):-
        append(T,L2,Rem_L3),
        =(L3,[H|Rem_L3]),
        =(L1,[H|T]).

     Clause 2:
     *********

  append(L1,L2,L3):-
        =(L1,[]),
        =(L2,L3).




     Analysis of clause 1:
     *********************

  append(L1,L2,L3):-
        append(T,L2,Rem_L3),
        =(L3,[H|Rem_L3]),
        =(L1,[H|T]).


Directionality 1:
*****************

   L1/$groundlist(1),
   L2/$groundlist(2),
   L3/$any(1)
     ::
   L1/$groundlist(1),
   L2/$groundlist(2),
   L3/$groundlist(3).

            ************************************


Permutation 1:
**************

  append(L1,L2,L3):-
        =(L3,[H|Rem_L3]),
        =(L1,[H|T]),
        append(T,L2,Rem_L3).


Permutation 2:
**************

  append(L1,L2,L3):-
        =(L1,[H|T]),
        =(L3,[H|Rem_L3]),
        append(T,L2,Rem_L3).


Permutation 3:
**************

  append(L1,L2,L3):-
        =(L1,[H|T]),
        append(T,L2,Rem_L3),
        =(L3,[H|Rem_L3]).


        (No more permutations for that clause!)



Directionality 2:
*****************

   L1/$any(1),
   L2/$any(2),
   L3/$groundlist(1)
     ::
   L1/$groundlist(1),
   L2/$groundlist(2),
   L3/$groundlist(3).

            ************************************


Permutation 1:
**************

  append(L1,L2,L3):-
        =(L3,[H|Rem_L3]),
        =(L1,[H|T]),
        append(T,L2,Rem_L3).


Permutation 2:
**************

  append(L1,L2,L3):-
```

```
        =(L3,[H|Rem_L3]),
        append(T,L2,Rem_L3),
        =(L1,[H|T]).


Permutation 3:
**************

  append(L1,L2,L3):-
        =(L1,[H|T]),
        =(L3,[H|Rem_L3]),
        append(T,L2,Rem_L3).


        (No more permutations for that clause!)
```

```
     Analysis of clause 2:
     *********************

  append(L1,L2,L3):-
        =(L1,[]),
        =(L2,L3).


Directionality 1:
*****************

   L1/$groundlist(1),
   L2/$groundlist(2),
   L3/$any(1)
     ::
   L1/$groundlist(1),
   L2/$groundlist(2),
   L3/$groundlist(3).

            ************************************


Permutation 1:
**************

  append(L1,L2,L3):-
        =(L1,[]),
        =(L2,L3).
```

```
Permutation 2:
**************

  append(L1,L2,L3):-
        =(L2,L3),
        =(L1,[]).


        (No more permutations for that clause!)


Directionality 2:
*****************

   L1/$any(1),
   L2/$any(2),
   L3/$groundlist(1)
     ::
   L1/$groundlist(1),
   L2/$groundlist(2),
   L3/$groundlist(3).

            ************************************


Permutation 1:
**************

  append(L1,L2,L3):-
        =(L1,[]),
        =(L2,L3).


Permutation 2:
**************

  append(L1,L2,L3):-
        =(L2,L3),
        =(L1,[]).


        (No more permutations for that clause!)
```

****
End of the analysis
****

```
          ********************************************
           ***Results of the syntactical transformation***
           ********************************************


     Clause 1:
     *********

  append(L1,L2,L3):-
        append(T,L2,Rem_L3),
        =(L3,[H|Rem_L3]),
        =(L1,[H|T]).

     Clause 2:
     *********

  append(L1,L2,L3):-
        =(L1,[]),
        =(L2,L3).




     Analysis of clause 1:
     *********************

  append(L1,L2,L3):-
        append(T,L2,Rem_L3),
        =(L3,[H|Rem_L3]),
        =(L1,[H|T]).


Common permutation 1:
*********************

  append(L1,L2,L3):-
        =(L3,[H|Rem_L3]),
        =(L1,[H|T]),
        append(T,L2,Rem_L3).


Common permutation 2:
*********************

  append(L1,L2,L3):-
        =(L1,[H|T]),
        =(L3,[H|Rem_L3]),
        append(T,L2,Rem_L3).


        (No more common permutations for that clause!)
```

```
     Analysis of clause 2:
     *********************

  append(L1,L2,L3):-
        =(L1,[]),
        =(L2,L3).


Common permutation 1:
*********************

  append(L1,L2,L3):-
        =(L1,[]),
        =(L2,L3).


Common permutation 2:
*********************

  append(L1,L2,L3):-
        =(L2,L3),
        =(L1,[]).


        (No more common permutations for that clause!)



**************************************************
           ***End of the analysis***
**************************************************
```

6. CODE DE LA PROCEDURE ET

TESTS EFFECTUES SUR MEILLEUR_PRIX/4.

```
{*********************************************
This DBASE module implements basic operations for handling tables (a set of
BIM prolog terms). So you can load a file into a table, save a table into a
file, put a term in the table and get/delete a set of terms respecting some
condition(s) from the table.

       by Pierre De Boeck, 16/07/92
*********************************************}



:- module(DBASE).

:-global DB__open_table/2 .
:-global DB__close_table/1 .
:-global DB__save_table/1 .
:-global DB__get_table/3 .
:-global DB__put_table/2 .
:-global DB__delete_table/2 .

{***********************
DB__open_table(atom _TB,_F)

in(gr,gr):out(gr,gr)<1,1>

precond: _F must be the complete UNIX_physical name of an existing file,
         containing BIM prolog terms
         No table _TB can exist

A table of name _TB is created and loaded with the content of _F.
_F is called the associated file to _TB.
***************************}
DB__open_table(_TB,_F):-
        fopen(_Pt,_F,'r'),
        read_lterm(_Pt,_L),
        rerecord(_TB,table(_F,_L)),
        fclose(_Pt).



{***********************
DB__close_table(atom _TB)

in(gr):out(gr)<1,1>

precond: a table _TB must exist

The content in _TB is saved in its associated file, and _TB is destroyed.
***************************}
DB__close_table(_TB):-
        recorded(_TB,table(_F,_L)),
        fopen(_Pt,_F,'w'),
        write_lterm(_Pt,_L),
        erase(_TB),
        fclose(_Pt).



{***********************
DB__save_table(atom _TB)

in(gr):out(gr)<1,1>

precond: a table _TB must exist

The content in _TB is saved in its associated file (_TB is not destroyed).
***************************}
DB__save_table(_TB):-
        recorded(_TB,table(_F,_L)),
        fopen(_Pt,_F,'w'),
        write_lterm(_Pt,_L),
        fclose(_Pt).



{***********************
DB__get_table(atom _TB; term _T; term_list _LT)

in(gr,_,var):out(gr,_,gr)<1,1>

precond: a table _TB must exist


_LT is the list of terms t:
        - t in _TB
        - t is an instance of _T
***************************}
DB__get_table(_TB,_T,_LT):-
        recorded(_TB,table(_F,_L)),
        select(_L,_T,_Ll,_LT).



{***********************
DB__put_table(atom _TB; term _T)

in(gr,gr):out(gr,gr)<1,1>

precond: a table _TB must exist


_TB+=_T
***************************}
DB__put_table(_TB,_T):-
        recorded(_TB,table(_F,_L)),
        rerecord(_TB,table(_F,[_T|_L])).



{***********************
DB__delete_table(atom _TB; term _T)
```

```
in(gr,_):out(gr,_)<1,1>

precond: a table _TB must exist


_TB-=t, forall t:
        - t in _TB
        - t is an instance of _T
**************************}
DB__delete_table(_TB,_T):-
        recorded(_TB,table(_F,_L)),
        select(_L,_T,_L1,_LT),
        rerecord(_TB,table(_F,_L1)).




{*****************  internal operations ***************************}



{***********************
read_lterm(term_FILE *_Pt; term_list _L)

in(gr,var):out(gr,gr)<1,1>

precond: *_Pt is opened in 'r'

All the terms in *_Pt (from the current head position) are read and stored in
_L. The head position is updated accordingly.
**************************}
read_lterm(_Pt,[_T|_L]):-
        read(_Pt,_T),!,
        read_lterm(_Pt,_L).
read_lterm(_Pt,[]).



{***********************
write_lterm(term_FILE *_Pt; term_list _L)

in(gr,gr):out(gr,gr)<1,1>

precond: *_Pt is opened in 'w'

All the terms in _L are written in *_Pt (from the current head position). The
head position is updated accordingly.
**************************}
write_lterm(_Pt,[]):-!.
write_lterm(_Pt,[_T|_L]):-
        writeq(_Pt,_T),
        write(_Pt,'.\n'),
        write_lterm(_Pt,_L).



{***********************
select(term_list _L; term _T; term_list _L1,_LT)

in(gr,_,var,var):out(gr,_,gr,gr)<1,1>

_LT is the list of terms t:
        - t in _L
        - t is an instance of _T
_L1 is _L \ _LT
**************************}
select([],_,[],[]):-!.
select([_T1|_L],_T,_L1,[_T1|_LT]):-
        copy(_T,_Tc),
        _T1=_T,!,
        select(_L,_Tc,_L1,_LT).
select([_T1|_L],_T,[_T1|_L1],_LT):-
        select(_L,_T,_L1,_LT).
```

```
query(_P,_PV,_F):-
        DB__open_table(CATA,'/home/patate/users/folon/bgb/TABLE/cata.tb'),
        DB__get_table(CATA,_,_CAT),
        DB__close_table(CATA),
        meilleur_prix(_P,_PV,_F,_CAT),
        write((meilleur_prix(_P,_PV,_F),'\n')),
        fail.
query(_P,_PV,_F).

{********************}

meilleur_prix(_P,_PV,_F,[cata_e(_P,_PV,_F)|_CAT_FT]):-
        test_mp(_P,_PV,_CAT_FT).
meilleur_prix(_P,_PV,_F,[cata_e(_P1,_PV1,_F1)|_CAT_FT]):-
        meilleur_prix(_P,_PV,_F,_CAT_FT),
        meilleur_prix_1(_P,_P1,_PV,_PV1,_F).

{********************}

meilleur_prix_1(_P,_P,_PV,_PV1,_F):-
        geq(_PV1,_PV).
meilleur_prix_1(_P,_P1,_PV,_PV1,_F):-
        not(_P=_P1).

{********************}

test_mp(_P,_PV,[]).
test_mp(_P,_PV,[cata_e(_P1,_PV1,_F1)|_CAT_FT]):-
        test_mp(_P,_PV,_CAT_FT),
        test_mp_1(_P,_P1,_PV,_PV1).

{********************}

test_mp_1(_P,_P,_PV,_PV1):-
        geq(_PV1,_PV).
test_mp_1(_P,_P1,_,_):-
        not(_P=_P1).

{********************}

geq(_PV1,_PV):-
        _PV1>=_PV.
```

```
cata_e(10,1500,1).
cata_e(20,2000,2).
cata_e(30,1800,3).
cata_e(10,1000,3).
cata_e(40,850,2).
cata_e(50,1750,1).
cata_e(40,700,1).
cata_e(20,1800,1).
cata_e(50,1800,3).
cata_e(20,1950,3).
cata_e(10,1000,2).
cata_e(10,1500,1).
cata_e(20,2000,2).
cata_e(30,1800,3).
cata_e(10,1000,3).
cata_e(40,850,2).
cata_e(50,1750,1).
cata_e(40,700,1).
cata_e(20,1800,1).
cata_e(50,1800,3).
cata_e(20,1950,3).
cata_e(10,1000,2).
cata_e(10,1500,1).
cata_e(20,2000,2).
cata_e(30,1800,3).
cata_e(10,1000,3).
cata_e(40,850,2).
cata_e(50,1750,1).
cata_e(40,700,1).
cata_e(20,1800,1).
cata_e(50,1800,3).
cata_e(20,1950,3).
cata_e(10,1000,2).
cata_e(10,1500,1).
cata_e(20,2000,2).
cata_e(30,1800,3).
cata_e(10,1000,3).
cata_e(40,850,2).
cata_e(50,1750,1).
cata_e(40,700,1).
cata_e(20,1800,1).
cata_e(50,1800,3).
cata_e(20,1950,3).
cata_e(10,1000,2).
cata_e(10,1500,1).
cata_e(20,2000,2).
cata_e(30,1800,3).
cata_e(10,1000,3).
cata_e(40,850,2).
cata_e(50,1750,1).
cata_e(40,700,1).
cata_e(20,1800,1).
cata_e(50,1800,3).
cata_e(20,1950,3).
cata_e(10,1000,2).
```

```
          ********************************************
           ***Results of the syntactical transformation***
           ********************************************


     Clause 1:
     *********

  meilleur_prix(P,PV,F,CAT):-
        not(=(P1,P)),
        meilleur_prix(P,PV,F,CAT_FT),
        =(CAT,[cata_e(P1,PV1,F1)|CAT_FT]).

     Clause 2:
     *********

  meilleur_prix(P,PV,F,CAT):-
        geq(PV1,PV),
        meilleur_prix(P,PV,F,CAT_FT),
        =(CAT,[cata_e(P,PV1,F1)|CAT_FT]).

     Clause 3:
     *********

  meilleur_prix(P,PV,F,CAT):-
        =(CAT,[cata_e(P,PV,F)|CAT_FT]),
        test_mp(P,PV,CAT_FT).




     Analysis of clause 1:
     *********************

  meilleur_prix(P,PV,F,CAT):-
        not(=(P1,P)),
        meilleur_prix(P,PV,F,CAT_FT),
        =(CAT,[cata_e(P1,PV1,F1)|CAT_FT]).


Common permutation 1:
*********************

  meilleur_prix(P,PV,F,CAT):-
        =(CAT,[cata_e(P1,PV1,F1)|CAT_FT]),
        meilleur_prix(P,PV,F,CAT_FT),
        not(=(P1,P)).


        (No more common permutations for that clause!)
```

```
     Analysis of clause 2:
     *********************

  meilleur_prix(P,PV,F,CAT):-
        geq(PV1,PV),
        meilleur_prix(P,PV,F,CAT_FT),
        =(CAT,[cata_e(P,PV1,F1)|CAT_FT]).


        (There is no common permutations for that clause!)




     Analysis of clause 3:
     *********************

  meilleur_prix(P,PV,F,CAT):-
        =(CAT,[cata_e(P,PV,F)|CAT_FT]),
        test_mp(P,PV,CAT_FT).


Common permutation 1:
*********************

  meilleur_prix(P,PV,F,CAT):-
        =(CAT,[cata_e(P,PV,F)|CAT_FT]),
        test_mp(P,PV,CAT_FT).


        (No more common permutations for that clause!)



*************************************************
          ***End of the analysis***
*************************************************
```

```
          ***********************************************
          ***Results of the syntactical transformation***
          ***********************************************


    Clause 1:
    *********

meilleur_pr3(P,PV,F,CAT):-
      not(=(P1,P)),
      meilleur_pr3(P,PV,F,CAT_FT),
      =(CAT,[cata_e(P1,PV1,F1)|CAT_FT]).

    Clause 2:
    *********

meilleur_pr3(P,PV,F,CAT):-
      geq(PV1,PV),
      meilleur_pr3(P,PV,F,CAT_FT),
      =(CAT,[cata_e(P,PV1,F1)|CAT_FT]).

    Clause 3:
    *********

meilleur_pr3(P,PV,F,CAT):-
      =(CAT,[cata_e(P,PV,F)|CAT_FT]),
      test_mp3(P,PV,CAT_FT).




    Analysis of clause 1:
    *********************

meilleur_pr3(P,PV,F,CAT):-
      not(=(P1,P)),
      meilleur_pr3(P,PV,F,CAT_FT),
      =(CAT,[cata_e(P1,PV1,F1)|CAT_FT]).


Common permutation 1:
*********************

meilleur_pr3(P,PV,F,CAT):-
      =(CAT,[cata_e(P1,PV1,F1)|CAT_FT]),
      meilleur_pr3(P,PV,F,CAT_FT),
      not(=(P1,P)).


      (No more common permutations for that clause!)


    Analysis of clause 2:
    *********************

meilleur_pr3(P,PV,F,CAT):-
      geq(PV1,PV),
      meilleur_pr3(P,PV,F,CAT_FT),
      =(CAT,[cata_e(P,PV1,F1)|CAT_FT]).


Common permutation 1:
*********************

meilleur_pr3(P,PV,F,CAT):-
      =(CAT,[cata_e(P,PV1,F1)|CAT_FT]),
      meilleur_pr3(P,PV,F,CAT_FT),
      geq(PV1,PV).


      (No more common permutations for that clause!)


    Analysis of clause 3:
    *********************

meilleur_pr3(P,PV,F,CAT):-
      =(CAT,[cata_e(P,PV,F)|CAT_FT]),
      test_mp3(P,PV,CAT_FT).


Common permutation 1:
*********************

meilleur_pr3(P,PV,F,CAT):-
      =(CAT,[cata_e(P,PV,F)|CAT_FT]),
      test_mp3(P,PV,CAT_FT).


      (No more common permutations for that clause!)



***************************************************
          ***End of the analysis***
***************************************************
```